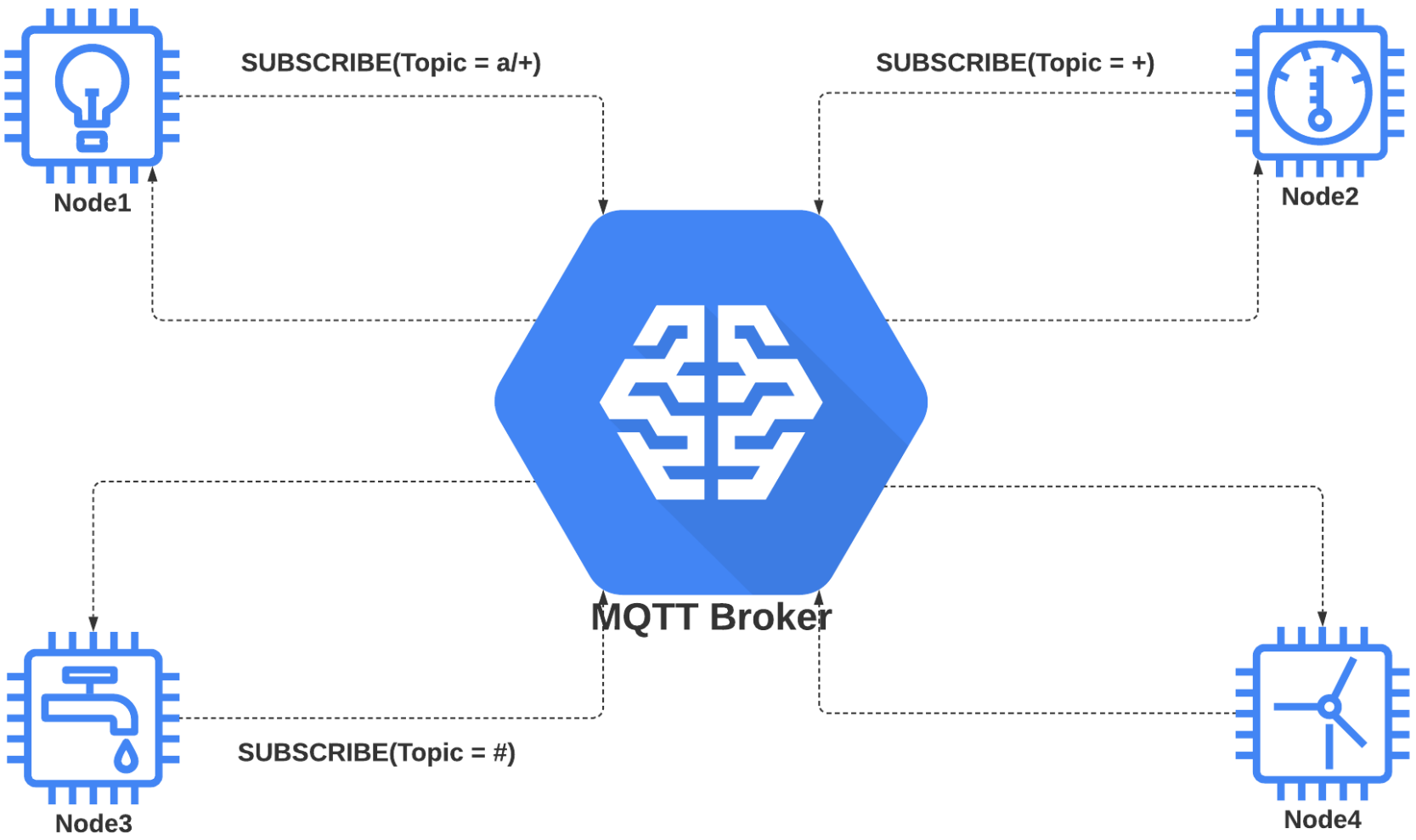
Payatu

IoT Security - Part 10 (Introduction To MQTT Protocol And Security)

Aseem

26-June-2020



This blog is part of IoT Security series where we discuss the basic concepts pertaining to the IoT/IIoT eco-system and its security. If you have not gone through the previous blogs in the series, I would urge you to go through those first. In case you are only interested in MQTT security, feel free to continue.

IoT Security - Part 1 (101 - IoT Introduction And Architecture)

IoT Security – Part 9 (Introduction To Software Defined Radio)

In this blog, we are going to look at one of the most famous and widely used IoT protocols – MQTT, security issues, and attacks on MQTT. But before that lets first get an idea of what exactly is MQTT and why is it the apple of IoT vendors' eye. Please note that we will use the terms Client and Node as well as Server and Broker interchangeably to mean the same thing.

1 Introduction

MQTT is short for Message Queueing Telemetry Transport. It is an OASIS and ISO Standard. It is based on the principle of client/Server publish/subscribe message mechanism. As with other IoT protocols, it is lightweight and simple to use and implement which works well for constrained and automated environments like M2M (Machine-to-machine) and IoT. It is an application layer protocol that runs over TCP and can run over any other network protocol that provides reliable, ordered, and bi-directional communication (e.g. websockets).

TCP Port	Communication Type
1883	Plain text
8883	TLS

The latest version of the protocol at the time of writing this blog is *5.0*. The protocol specification can be found here - https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

1.1 Use and Popularity

MQTT has gained a lot of popularity in the Industrial Internet of Things (IIoT) eco-systems, Industrial Control Systems (ICS), Enterprise IoT, etc. given its simplicity, flexible architecture, and cloud readiness. We think MQTT is going to play an extremely vital role in Industrial Control Systems (ICS) and IIoT infrastructures in the near future.

1.2 Architecture Overview

At the center of the MQTT network is an MQTT broker. All the MQTT nodes connect to the broker. Think of it as a star network. A node can subscribe for as well as publish messages. The nodes communicate with each other via the broker by publishing or subscribing to the broker. In simple terms, the broker forwards the messages from one node to the others based on subscriptions.

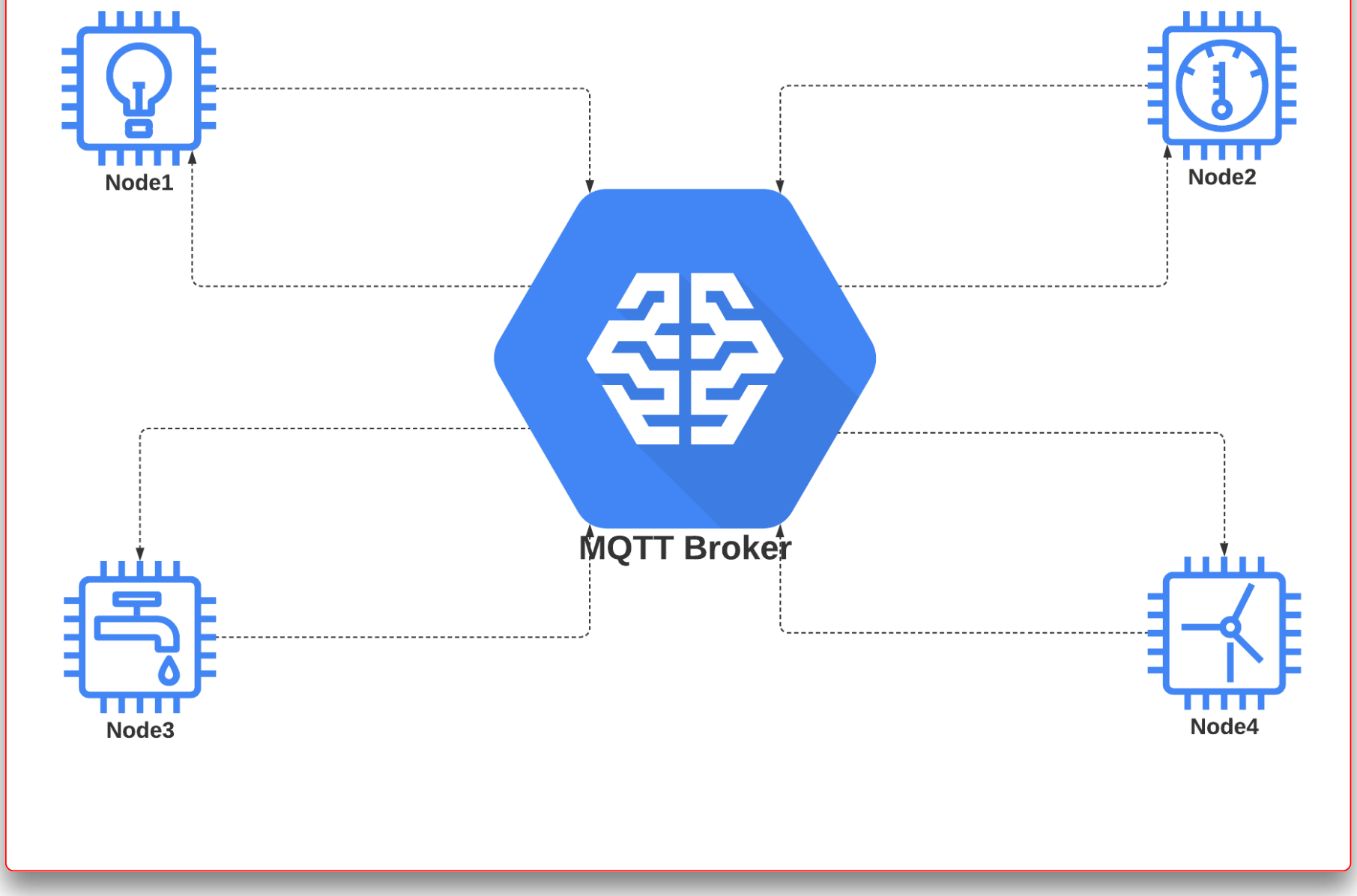


Fig 1. A simplified view of the MQTT network

What components of the IoT eco-system will act as a broker and the nodes are totally dependent on the requirements of the Infrastructure, eco-system, and implementation.

A node can be (but not limited to):

1. An IoT edge device
2. An IoT Gateway
3. An application Server
4. Mobile Application
5. Cloud Application
6. Another MQTT broker (when bridging)

Similarly, a broker can be (but not limited to):

1. An IoT Gateway
2. A Cloud-based MQTT server
3. Mobile Application

2 MQTT Communication

2.1 Communication Overview

So, how do nodes specify whom to send messages to and to accept messages from? That is done using something called *Topic*. The nodes basically publish a message on a topic and other nodes can specify to the broker which topics they want to subscribe to. It's probably time to define

some terminology and explanation about topics

1. Topic - A topic or topic name is just a label that is attached to a Message (Application payload). The syntax of a topic name is similar to a tree structure (Think file system tree, without a starting /). A valid topic for example could be payatu/office1/boardroom/temperature.
2. Topic Level - Each item in a topic represents a topic level and the topic levels have a parent-child relationship, for e.g. in payatu/office1/boardroom/temperature *office1* is at a parent topic level of *boardroom*.
3. Topic level separator - Each topic level is separated by a forward slash /.
4. Topic Filter - A node can subscribe to one or more topics in a single request to an MQTT broker by specifying a topic filter. A topic filter is basically an expression that matches one or more topics (think simple regex). There are two wildcards that can be used to match multiple topics.
    1. Single-level wildcard + - is used to match any item in a specific topic level. Let's look at some examples to make it clear:
        1. a/+/c/d topic filter will match a/b/c/d , a/x/c/d but not a/b/y/d.
        2. a/b/c/+ topic filter will match a/b/c/d and a/b/c/x but not a/x/c/d or a/b/c/x/y
    2. Multi-level wildcard # - It is used to match any item at the specific topic level and all its child topic levels. It must be specified at the leaf of the topic name. Some examples for # wildcard:
        1. a/b/# topic filter will match any topic level under "b" and any of its children i.e. a/b/c, a/b/c/d. It will not match a/x/c or a/b
    3. Both the wildcards can be used in a single topic filter e.g. a/+/c/#
    4. Single wildcard character filters are also valid for example # and +
5. Topic starting with $ character - The topic names starting with the *$* character (for e.g. *$foo/bar/boo*) are generally used for MQTT server implementation and management purposes. $SYS/ is a widely-used topic prefix for server-specific information or control APIs as per the spec. However, this is not a mandate. Server implementations are free to use any prefix of their choice for e.g. AWS IoT uses the topic prefix $aws/ for all internal/management stuff.

   There are certain conditions and restrictions applied to these topics:
    1. The MQTT server will not match the topic filters starting with wildcard # or + with topic names starting with *$* character.
    2. The nodes should not use topic names starting with *$* character for exchanging messages with other nodes. In other words, they should not be used for application-specific messages.

2.2 Communication Example

The below example demonstrates a scenario where we have 4 nodes connected to an MQTT broker. Three nodes (Node1, Node2, and Node3) subscribe to topics of their choices using different topic filters. One node (Node1) then publishes a message on a topic to the broker. Out of the three subscribing nodes, the broker publishes the message to only two nodes (Node2, and Node3) as their topic filters match the topic on which the message is published.

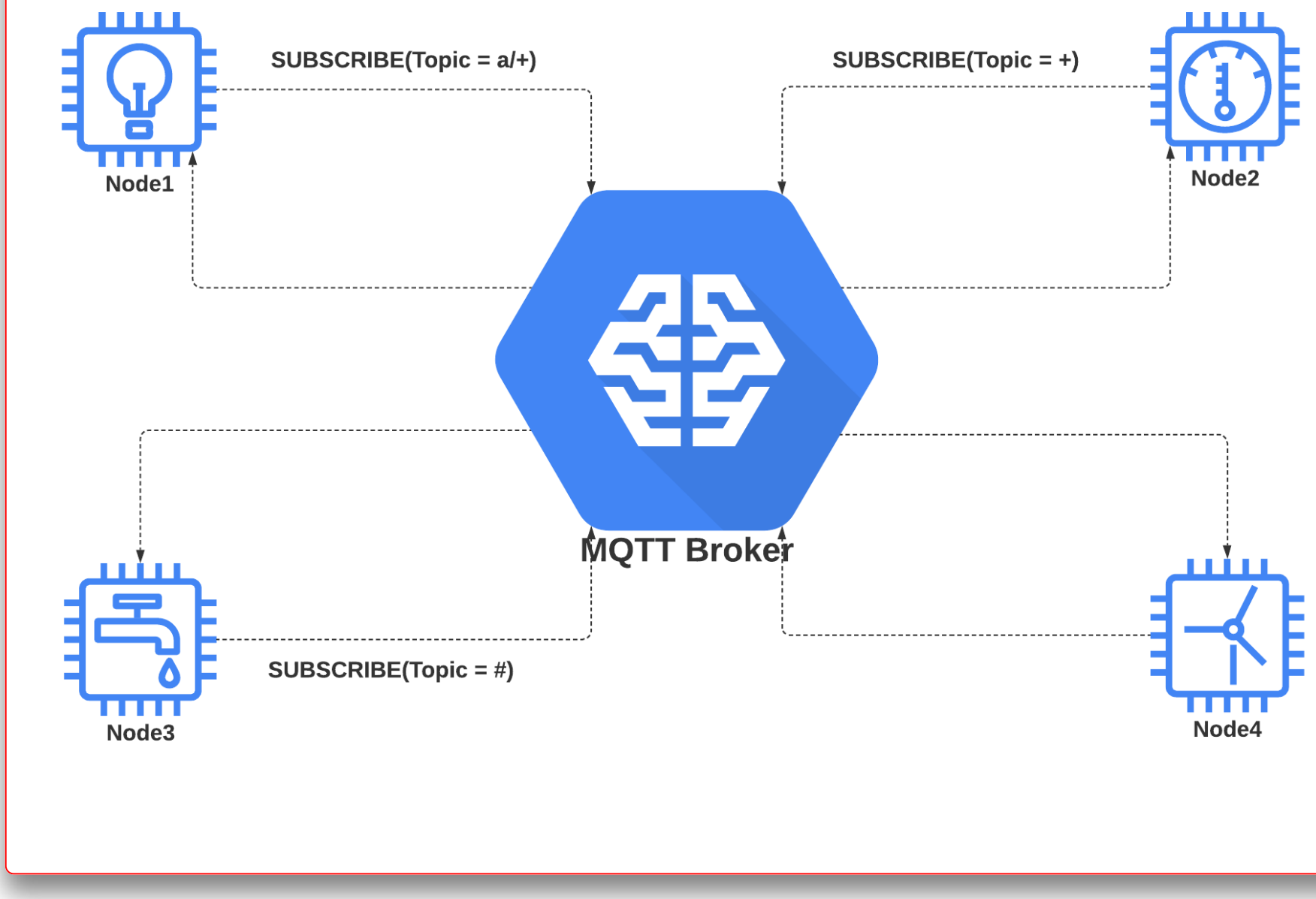


Fig 2. Three nodes subscribe to different topic filters

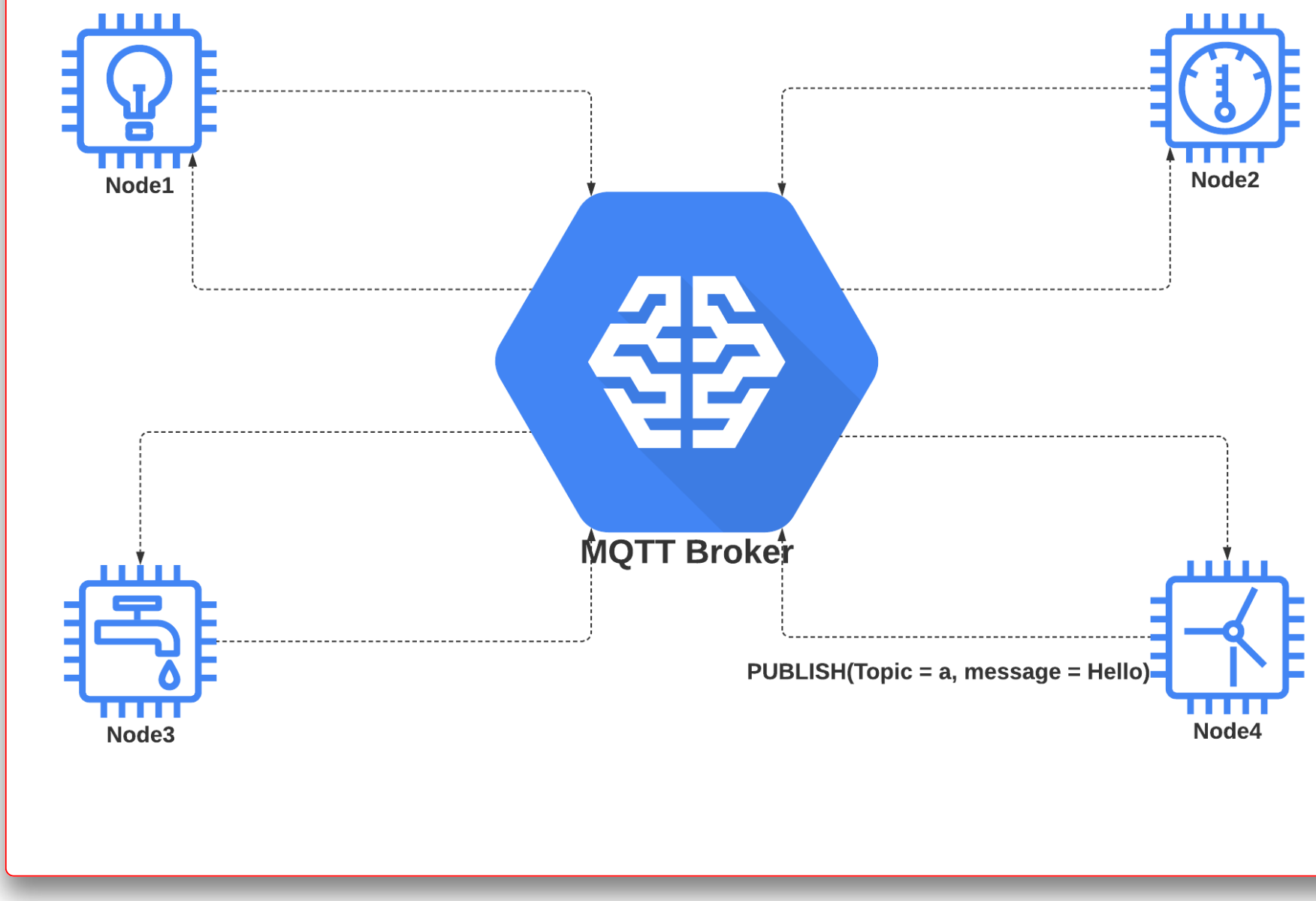


Fig 3. One node publishes a message “Hello” on topic “a

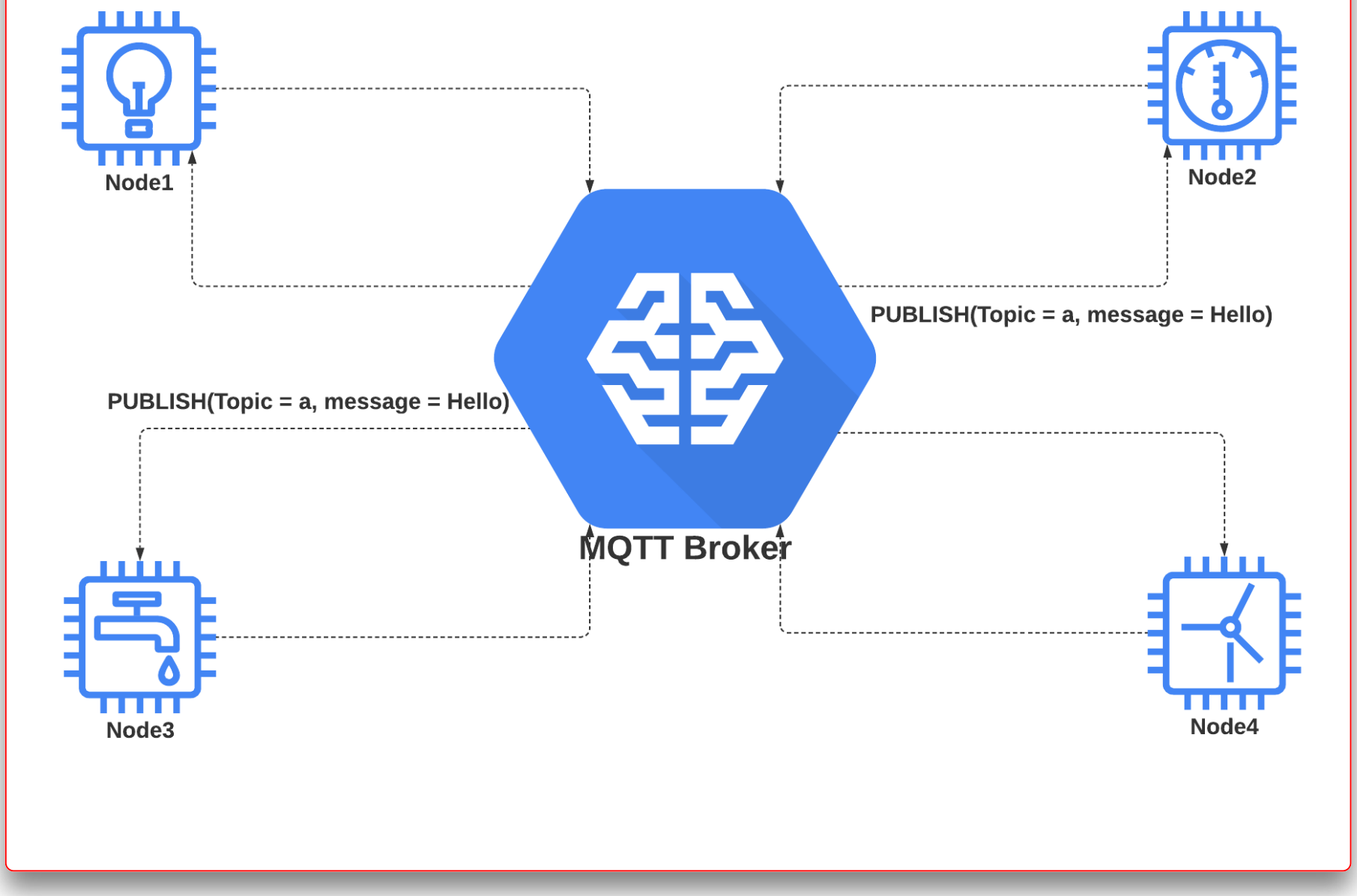


Fig 4. Two nodes receive the message “Hello” on the topic “a” because their subscription topic filters match the topic “a”

Assuming the same setup as above, If Node1 published messages on the below-mentioned topics to the broker, the nodes that will receive the messages from the broker for each topic are -

Topic on which message is published	Nodes That will receive the Message
a/b	Node1 and Node3
a/b/c	Node3
$SYS/broker/foo	None

3 MQTT Protocol Intro

Since this blog focuses on the introduction, we will not go into the protocol internal details. However, we will discuss the important aspects of the protocol and communication here and may write a detailed blog on MQTT Protocol internals if required.

3.1 MQTT Control Packets Intro

The communication between the client (nodes) and the server (broker) happens over MQTT Control Packets. An MQTT control packet consists of one to three sections:

1. Fixed Header - This is present in all packets. This specifies the control packet type, flags, and the remaining packet length
2. Variable Header - This is present in some packets. The contents of this header vary depending on the packet type. Usually, Packet Identifier is one of the fields in some packet types.

3. Payload - This is present in some packets like CONNECT, PUBLISH, SUBSCRIBE, SUBACK, UNSUBSCRIBE, and UNSUBACK. In the PUBLISH packet, the payload is the application message and is surprisingly optional.

There are 16 MQTT control Packets defined in the MQTT version 5.0 specification.

Packet Type	Description
Reserved	Reserved
CONNECT	Connection Request
CONNACK	Connection Acknowledgement
PUBLISH	Publish a message
PUBACK	Publish Acknowledgement (QoS = 1)
PUBREC	Publish Received (QoS = 2)
PUBREL	Publish Release (QoS = 2)
PUBCOMP	Publish Complete (QoS = 2)
SUBSCRIBE	Subscribe Request
SUBACK	Subscribe Acknowledgement
UNSUBSCRIBE	Unsubscribe Request
UNSUBACK	Unsubscribe Acknowledgement
PINGREQ	Ping Request
PINGRESP	Ping response
DISCONNECT	Disconnect notification
AUTH	Authentication exchange

3.2 Quality of Service (QoS)

The protocol defines three levels of communication reliability via the *Quality of Service* flag. The value of this flag determines how the communication between the sender and the receiver will take place i.e. using different control packets. We mentioned the terms sender and receiver here because both client (node) and server (broker) can act as either sender or receiver depending on who is sending the message to whom for e.g. it could be from node to broker or from broker to node. Let's look at the three levels

1. QoS level 0 - At most once delivery- This is an unreliable messaging mechanism where no acknowledgment is sent by the receiver which means if the message is lost, it's lost :).

Sender	Flow	Receiver
Sends PUBLISH with QoS = 0. Transaction complete	—->	Receives the message (or not)

1. QoS level 1 - At least once delivery - This ensures that the message is received at least once. The receiver is supposed to send the PUBACK packet in response to a PUBLISH packet.

Sender	Flow	Receiver

Sender	Flow	Receiver
Sends PUBLISH with QoS = 1	—->	Receives the message
Receives the PUBACK. Discards message. Transaction complete.	<—-	Sends PUBACK

1. QoS level 2 - Exactly once delivery - This is used when loss or duplication of the message is not acceptable. It is a 4-step process.

Sender	Flow	Receiver
Sends PUBLISH with QoS = 1	—->	Stores the Packet ID. May Initiate onward delivery.
Discards message. Stores PUBREC with Packet ID	<—-	Sends PUBREC with the same Packet ID
Sends PUBREL with same Packet ID	—->	Discards Packet ID for any future packets received
Discards stored state. Transaction complete	<—-	Sends PUBCOMP with the same Packet ID

3.3 Packets and Usage

We will discuss some of the actions (and respective packets, their contents) briefly as describing all control packets along with each field would be an overkill for this 101-style blog. However, we would urge you to go through the protocol specification to get a good understanding of the internals.

3.3.1 CONNECT

This is a logical MQTT connection request. This is the first MQTT control packet that is sent from the client to the server after the underlying connection is established. It contains some important information for the connection. Some of the important info in the variable header of the packet includes:

1. Protocol - It includes protocol name and version
2. Connect Flags - One byte specifying the corresponding values being present in the payload or not, things like user name, password, *Will* flag, Clean start, etc. The 'Will' stuff is basically used by the client to tell the server to send a specific (*will*) message on the *will* topic and utilize other *will* data if the client disconnects abruptly.
3. Keep Alive - Time in seconds between two packet transmission, if elapsed the client must send a PINGREQ packet.
4. Properties - Some packets, including CONNECT, contain standard properties, defined in the spec, at the end of the variable header.

Some of the important information in the payload includes:

1. Client Identifier - Each client connecting to the server specifies a unique client ID. This is used for maintaining the session state. Note that this is not part of the MQTT authentication.
2. User Name - The payload contains the user name if the user name flag is set in the variable header
3. Password - The actual password, if the password flag is set in the variable header.

3.3.2 CONNACK

This packet is the response for the CONNECT packet and contains information about the status of the connection request. It can either accept or reject the connection. The variable header contains the Connect Reason Code, connection Acknowledgement Flags, and Properties. The connect reason code is a single byte that specifies success, failure, errors of the connection request such as 'Bad User Name or Password', 'Protocol Error', 'Success', 'Malformed Packet' to name a few. There are 22 Connect Reason Codes defined in the specification.

3.3.3 PUBLISH

This is used to publish an Application Message on a topic. The variable header includes:

1. Topic Name - The topic to publish the message on.
2. Packet Identifier - Present only for packets with QoS = 1 or 2 as it is required to send the respective acknowledgment packets.
3. Properties - Publish specific properties. Check the specification for more details

The payload of the packet contains the Application Message being published. The format of the data in the payload is application-specific. So, developers are free to decide what kind of data they want to exchange such as JSON, XML, text, binary, etc. For e.g. AWS IoT uses JSON format for data exchange between AWS things and IoT core.

The response to PUBLISH depends on the QoS

1. QoS = 0 - No response
2. QoS = 1 - PUBACK
3. QoS = 2 - PUBREC (followed by an exchange of PUBREL and PUBCOMP)

3.3.4 SUBSCRIBE

It is used to subscribe to one or more topics for receiving Application Messages being passed on the same. The variable header contains a *Packet Identifier* and *Properties*. The payload contains a list of Topic filters and each topic filter is followed by a byte describing its subscription options.

3.3.5 SUBACK

This packet is the response for the SUBSCRIBE packet and contains information about the receipt and the processing of the subscribe request. It can selectively grant or reject subscription requests for topic filters based on any issues. The variable header contains the packet identifier and Properties. The payload contains a list of Subscribe Reason Codes where each reason code corresponds to the respective Topic filter in the subscribe request i.e. the order of reason code is the same as the order of topic filters in the subscribe request. A reason code is a single byte that specifies success, failure, errors corresponding to the respective topic filter such as 'Not Authorized', 'Topic Filter Invalid', 'Granted QoS 0', 'Wildcard subscriptions not supported' to name a few. There are 12 Subscribe Reason Codes defined in the specification.

3.3.6 UNSUBSCRIBE

This packet is used to unsubscribe from one or more topics. The variable header contains the packet identifier and Properties. The payload contains a list of topic filters that the client wants to unsubscribe from.

3.3.7 UNSUBACK

This packet is a response to an UNSUBSCRIBE packet to confirm the receipt of the unsubscribe request. The variable header contains the packet identifier and Properties. The payload contains a list of Unsubscribe Reason Codes where each reason code corresponds to the respective Topic filter in the unsubscribe request i.e. the order of reason code is the same as the order of topic filters in the unsubscribe request. A reason code is a single byte that specifies success, failure, errors corresponding to the respective topic filter such as 'Success', 'Not Authorized', 'Topic Filter Invalid' to name a few. There are 7 Unsubscribe Reason Codes defined in the specification.

The other packets include:

1. PINGREQ and PINGRESP are used for connection keep-alive or heartbeats
2. AUTH is used when an extended or external authentication mechanism is supposed to be used in the MQTT session.
3. DISCONNECT to close the MQTT session.

4 MQTT Security

Now that we understand how MQTT works, let's focus on the security aspect of MQTT both from an offensive as well as defensive perspective. All the information below is based on our experience with MQTT security research and IoT product and infrastructure penetration testing projects. We will try to keep things simple in this blog. In future blog posts, we will dig deeper into the technical aspects of the different attacks on MQTT. Below we will try to answer some of the questions that generally come up when one encounters a new protocol in their IoT penetration tests.

4.1 Tools

Are there any tools available that assist in MQTT analysis and assessment? Thankfully there are some choices available:

1. Open-source libraries, clients, server implementations. The only issue here is that they are rigid and have no guidance from the security perspective and you may end up editing or writing your own scripts to do the job.
2. **EXPLIoT Framework** has some pretty neat MQTT plugins for *publish*, *subscribe* and even MQTT Authentication cracking. It can interact with standard MQTT servers as well as AWS IoT MQTT servers.
3. Fuzzing - Scapy has MQTT packet creation capabilities and this can be used to create your own fuzzers.

4.2 Recon

Ok, so we have the tools, now what? Where do we start? The first thing is to perform recon on the MQTT broker get as much information about the MQTT network as possible.

1. Can you connect without authentication?
2. What happens when you subscribe to all topics i.e. `#`
3. Do you have access to any of the`$SYS/` topics? What is the information available?

4. Are you able to read any sensitive information?
5. Are you able to publish to any or sensitive topics?
6. What topics are important and used in the IoT Eco-system? This can also be found by reverse-engineering the device firmware or Mobile App (if it is part of the MQTT network).
7. How are Client ID generated? Is there any pattern to the client IDs?
8. Is TLS used?
9. How is the telemetry data sent from the device utilized on the cloud?

4.3 Analysis

Once we gather the basic information about the MQTT implementation, we can utilize it to launch different attacks on the eco-system. By eco-system, I mean all the components of the MQTT infrastructure including the devices, gateways, Cloud Broker and apps, mobile apps, etc. We will list down some of the ideas to get your grey cells working:

1. Subscribing to # helps you understand which topics you have access to with or without authentication as well as the messages that are passed.
2. If the Broker requires authentication, you can launch password brute-force or dictionary-based attacks
3. Reversing the firmware or the mobile app may give you sensitive information such as
    1. Client IDs
    2. Credentials
    3. Certificates and keys
    4. Topics used
    5. What data is exchanged and expected on specific topics and how it affects the behavior of the components.

4.4 Attacks

4.4.1 Denial of Service via Duplicate Client ID

During our research, a few years ago, one of the things we found was that almost all MQTT implementations have the same behavior when it comes to duplicate Client IDs. The scenario is when a client connects to the server, if the client ID, presented by the new client that sent this CONNECT request, is already in use by another client that is currently connected to the server, the server goes ahead and disconnects the currently connected and lets the new client connect. There might be valid reasons for this behavior hence it is not treated as a bug. However, this issue has the potential of causing DoS within the MQTT network. The information that is required is the logic for generating client IDs and it becomes easy if they use a pattern. All you need in this case is a script that generates the client IDs and connects to the broker to effectively kick-out most if not all currently connected clients in other words cause an MQTT DDoS.

4.4.2 Client ID used for Authentication/Access Control

Sometimes (True story!) the client IDs might be used for authenticating a client or providing access to sensitive topics. In this case, using the same techniques as above, you can get access to the MQTT network and receive or manipulate sensitive data.

4.4.3 Cloning the Client

Again, depending on whether authentication is implemented or not and your access to credentials/keys of a legitimate thing, you can spoof the legitimate client and manipulate the telemetry data as if it was coming from the actual client or receive sensitive information destined for the legitimate client.

4.4.4 Attacking the Application via Malicious Input

Note: We have found and already disclosed a few vulnerabilities to known MQTT implementations using this technique. We are currently waiting for the disclosure timeline to finish so we can post a detailed blog on it.

The IoT protocols were created for constrained environments. They are relatively new and the portability and data exchange with conventional protocols and applications is handled by the applications unless the protocol standard specifies the exchange mechanism. This is not a complex problem at all, but this opens a new attack surface. The Application developers typically trust the telemetry data coming from the device as the data is minimal in nature and may forget to filter it for known attacks. This means that depending on how the data will be processed on the application side, if the telemetry data contains a maliciously crafted payload, it can exploit the application. Let's take a very simple example of an ICS/IIoT Infrastructure where the temperature of a certain room is regularly sent to the application and the application displays it on the web UI. Assuming, the attacker has access to the MQTT network and can publish to the same topic, now publishes an XSS payload, this is bound to exploit the browser if the temperature data is not filtered by the application.

Disclaimer: This technique is basically applicable to any IoT protocol where telemetry data is transferred from device to the cloud using the constrained IoT protocol (and maybe converting it to a conventional protocol like HTTP) and then fed to standard applications for storage, processing or display without first filtering it.

4.4.5 Accessing and Manipulating Devices via messages

Using various techniques mentioned above, if you can receive sensitive messages (by subscribing to interesting topics) and/or are able to publish messages of your choice to sensitive topics, it is game over for the MQTT network. An example of publishing a message on a topic could be controlling the device by sending it specific commands on the topic that it expects to receive commands from the Gateway/Cloud/User. That is scary!

4.5 Mitigations

Everything is not so bad with MQTT else nobody would be using it in the first place. The vulnerabilities and issues in MQTT as with any other protocol stem from insecure configuration and use of the implementations. Some of the things that you should look at as a base-line for securing your MQTT implementation are:

1. Use TLS
2. Don't use Client IDs for authentication or topic access control.

3. Use randomly generated Client IDs
4. Do not expose `$SYS/` topics to clients. If required use authentication based access control.
5. Treat Application Messages coming from the devices as tainted and filter them for known attacks based on the usage of that data.
6. Don't Hardcode credentials in the device firmware.
7. Devices (Clients) should have visibility or access to only the required topics.
8. Log all activity.
9. Prefer to log messages on `$SYS/` topics instead of publishing it on those topics and making it available for subscription, if it is not required. For e.g. Mosquitto server has a configuration option to log locally instead of publishing messages on any `$SYS/` topic.
10. Enable and use Cloud provider (AWS, Azure, etc.) security notifications for misbehavior or potential attacks in the MQTT infrastructure, for e.g. multiple failed Authentication attempts.

5 Conclusion

We hope this blog post gave you a good high-level overview of MQTT protocol, how it works, and how it is used and how would you go about performing security assessments of MQTT based IoT/IIoT eco-systems. The tools, attacks, and techniques mentioned will improve assessment efficiency and give you a direction for a deeper inspection of security research on any MQTT implementation.

Continue to the next part - IoT Security - Part 11 (Introduction To CoAP Protocol And Security)