SMOKER

Client-managed anonymous authentication and authorization for MQTT

Bachelor thesis

This thesis introduces a Proof of Concept implementing two independent security processes for MQTT 5.0: Anonymous authentication based on Zero-Knowledge Proofs as well as client-managed authorization. It provides an enhanced authentication scheme based on the Schnorr Non-Interactive Zero-Knowledge Proof over an elliptic curve. Therefore, no sensitive data has to be transmitted, nor stored on the broker. In case the broker gets compromised, no valuable data can be extracted (e.g. password hashes). Furthermore, the approach introduces a client-managed authorization mechanism usable for authenticated parties. Both authentication and authorization pursue the goal of not having a central authority and delegating the data sovereignty to the clients.

Degree course:	BSc in Computer Science
Authors:	Cédric von Allmen, Lukas Läderach
Tutor:	Dr. Reto Koenig
Experts:	Dr. Federico Flueckiger
Constituent:	Bern University of Applied Sciences
Date:	March 28, 2020

Management summary

This thesis introduces a Proof of Concept implementing two independent security processes for MQTT 5.0: Anonymous authentication based on Zero-Knowledge Proofs as well as client-managed authorization.

MQTT is a communication protocol widely used in the field of Internet of Things (IoT). It's a client-server messaging transport protocol, based on the publish/subscribe pattern. Since, the approach shown in this thesis mainly targets IoT devices, a certain efficiency needs to be taken into account.

Until version 3.1.1, only username and password were allowed to gain client authenticity, which, however, required the client to deliver his secret over a basically untrusted network channel (e.g. internet). This forces the client and the broker to set up channel and network security like TLS. Furthermore, if the username and password approach is used, the broker becomes responsible to securely store client secrets. However, this makes the broker a valuable target for attacks.

Since the MQTT specification 5.0 (as of March 2019), the *enhanced authentication* mechanism was introduced. This makes it possible to implement advanced authentication scenarios.

In this thesis an enhanced authentication scheme based on the Schnorr Non-Interactive Zero-Knowledge Proof over an elliptic curve is introduced. Therefore, no sensitive data has to be transmitted, nor stored on the broker. In case the broker gets compromised, no valuable data can be extracted (e.g. password hashes).

Furthermore, the approach introduces a client-managed authorization mechanism for authenticated parties. This mechanism lets the client claim topics, within a predefined client space. Only the client that owns the topic can finely grain the permissions. This allows the client to fully manage a topic and thus create his own chain of trust between several authenticated clients.

About this document

This documentation was written as part of the bachelor thesis for the degree "BSc Computer Science" at the Bern University of Applied Science.

Structure

The document is split into three parts, each representing a vital part of the thesis.

- Part 1: White Papers – A high level view of the idea behind the thesis and the basis for most features.
- Part 2: Implementation – Description of the the resulting Proof of Concept (PoC).
- Part 3: Methodology – In depth explanation, how we managed to achieve the results described in Part II.

Glossary

The glossary attached to this thesis, mostly consists of summaries taken from Wikipedia. Since those summaries are quite accurate, there is no reason to write them again.

Gender neutrality

The use of the masculine gender has been adopted for ease of reading and has no discriminatory intent.

Brainstorming During the work on this thesis, several discussions were started that ended up in a specific decision how something should be implemented, documented or named. Deriving solutions based on problems has often taken a lot of time – but the result has always been very valuable. Therefore, these decisions were documented with so called "Brainstorming" sections distributed all over the document, to point out our solution finding process.

Contents

1. Preliminaries

This chapter gives the reader a brief introduction about the technology and terms used in this thesis. This is the bare minimum which is required to further understand this document.

1.1. MQTT

Message Queuing Telemetry Transport (MQTT) is a Client Server publish/subscribe messaging transport protocol. It is lightweight, simple, and offers near real time communication abilities. These characteristics make it ideal for use in many situations, including constrained environments like Machine to Machine (M2M) and IoT contexts, where a small code footprint is required and/or network bandwidth is limited [1]. The central server instance (broker) is managing the topics which are the main resource to provide communication between clients. Figure 1.1 shows the basic network topology on which MQTT is based.

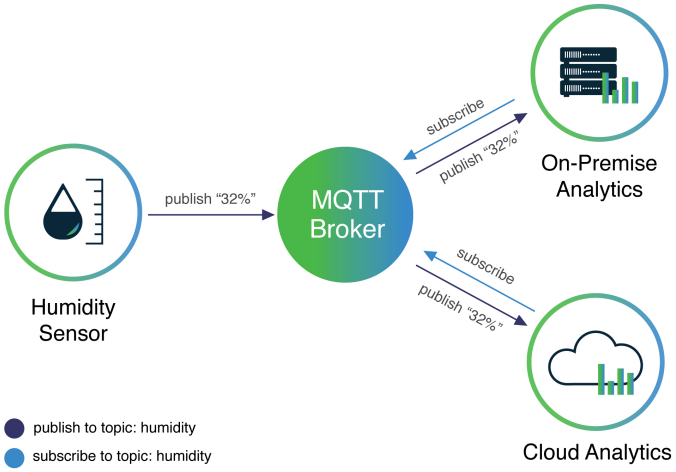


Figure 1.1.: MQTT network topolgy [2].

1.1.1. Topics

In MQTT, the word topic refers to an UTF-8 string that the broker uses to filter messages for each connected client. The topic consists of one or more topic levels. Each topic level is separated by a forward slash (topic level separator) [3]. Topic segments may also contain wildcard characters (e.g. +, #).

1.1.2. QoS

MQTT delivers application messages according to QoS levels. The following QoS levels are specified by the MQTT protocol [4, Section 4.3]:

- **QoS 0: At most once delivery**: The message is delivered according to the capabilities of the underlying network. No response is sent by the receiver and no retry is performed by the sender. The message arrives at the receiver either once or not at all.

- **QoS 1: At least once delivery**: This Quality of Service level ensures that the message arrives at the receiver at least once. A QoS 1 PUBLISH packet has a Packet Identifier in its Variable Header and is acknowledged by a PUBACK packet.
- **QoS 2: Exactly once delivery**: This is the highest Quality of Service level, for use when neither loss nor duplication of messages are acceptable. There is an increased overhead associated with QoS 2.

1.2. JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans and machines to read and write. It is based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition – December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript. These properties make JSON an ideal data-interchange language [5].

1.3. Public-key cryptography

The term public-key cryptography is used interchangeably with asymmetric cryptography. They both denote exactly the same thing and are used interchangeably. A public-key cryptographic system uses pairs of keys: *public keys* which may be spread widely, and private keys which are only known to the owner. The generation of such keys depends on cryptographic algorithms based on mathematical problems to produce one-way functions. Effective security only requires keeping the private key private; the public key can be openly distributed without compromising security. [6] [7]

1.4. Digital signatures

Digital signatures are one of the most important cryptographic tools – they are widely used today. Applications for digital signatures range from digital certificates for secure e-commerce to legal signing of contracts to secure software updates. Together with key establishment over insecure channels, they form the most important instance for public-key cryptography.

Digital signatures share some functionality with handwritten signatures. In particular, they provide a method to assure that a message is authentic to one user, i.e., it in fact originates from the person who claims to have generated the message. However, they actually provide much more functionality, which we won't cover in detail, though. More information is available in the excellent book Understanding cryptography [6].

1.5. Authentication

Authentication is the act of validating that users are, who they claim to be. Basically authentication answers the question *Who are you?* [8]

1.6. Authorization

Authorization is a process of giving a user permission to access a specific resource or function. Basically authorization answers the question *What are you allowed to do?* [9]

2. Trust setting

The trust setting defines which parties are trusted to simplify threat modeling. Attacks or weaknesses of trusted parties are out of scope and are not analyzed in the context of this thesis.

2.1. Trusted parties

Hosting environment The hosting environment reflects a system (usually an operating system), where the MQTT broker and possible other companion software is running. Any 3rd party software which is required to run the broker (e.g. Docker or .NET Core runtime), is fully trusted.

Source code The code produced in this thesis is fully trusted. It is assumed that only this code is running. The case that an attacker has gained access to the host system and injects his own modified payload, is therefore out of scope.

Client secrets We assume the client is able to keep his private key, which is used for authentication, hidden. Therefore, if a client proofs its authenticity, a full trust to this client is established.

2.2. Not trusted parties

External data sources The broker may use a database (e.g. Mongo database) which may even be hosted on an external provider. The case, in which an attacker gains not permitted access and manipulates data is an eminent threat. This thesis provides a solution to partially solve this issue.

Communication channel It must always be assumed that a Man In The Middle (MITM) is sitting between the client and the broker during the authentication process and is able to eves drop the data. Therefore, the communication channel between the client and the broker can not be trusted.

Part I.

White papers

3. How to Authenticate MQTT Sessions Without Channel- and Broker Security

This paper describes a new but state of the art approach to provide authenticity in MQTT sessions using the means of Zero-Knowledge Proofs. This approach completely voids session hijacking for the MQTT protocol and provides authenticity without the need for any network security nor channel security nor broker based predefined Access Control List (ACL). The presented approach does not require the broker to keep any secrets for session handling. Moreover, it allows the clientID, which represents the identification for a session, to be publicly known. The presented approach allows completely anonymous but authentic sessions, hence the broker does not need any priory knowledge of the client party. As it is especially targeted for applications within the world of IoT, the presented approach is designed to require only the minimum in extra power in terms of energy and space. The approach does not introduce any new concept, but simply combines a state of the art cryptographic Zero-Knowledge Proof of identity with the existing MQTT 5.0 specification. Thus, no protocol extension is required in order to provide the targeted security properties. The described approach is completely agnostic to the application layer at the client side and is only required during MQTT session establishment.

This paper was initially drafted and published [10] during project 2. Since the paper was only a draft, a more complete version was created and published during this thesis.

3.1. Introduction

MQTT is designed as a robust, session-oriented protocol especially suitable for the world of IoT, where the clientID plays the central role for session management. The MQTT specification requires the clientID to be provided within the first data frame of the protocol during session establishment. The semantics of the clientID is to provide the unique way a session can be (re)established between a client and the broker, without any further information. So the clientID is required to be unique per broker over time, hence, no collision of clientIDs should ever happen. As the clients are not aware of each other, but usually provide their own clientID, it must be drawn from large set of possible clientIDs so the probability of a collision of clientIDs is negligible. The protocol specification defines a minimum clientID space of order 63^{23}. Hence, allows the theoretical security parameter of 2^{λ} with an approximate $\lambda = 137$.

The specification of MQTT does not directly address the immanent possibility of active session hijacking. This attack vector allows an adversary to take over the session by simply (re-)establishing a connection using the clientID of the victim. Only indirectly, the MQTT specification tries to weaken that immanent attack vector by providing the possibility for channel security via Transport Layer Security (TLS) or even worse, by securing a whole network via Virtual Private Network (VPN), and of course the heavy security requirement of the broker to be fully trusted in terms of secrecy of the clientID. However, the MQTT protocol specification gives room for customized authentication and even provides protocol intrinsic authentication solution via simple username password or via ACLs for ID-based session handling. These ID-based approaches, however, pose several disadvantages in the context of client management and security, as they require a separate and out-of-band on-staging-phase comprising the broker in order to manage clients by the identification scheme. Moreover, all these methods require the broker to keep secrets shared amongst different clients and the broker.

The presented approach provides a fully anonymous identification scheme based on the Schnorr Non-Interactive Zero-Knowledge Proof (Schnorr NIZKP) identification scheme [11] without any of the drawbacks mentioned above. It makes use of well-known cryptographic properties and allows the security model of the broker to be lowered from "fully trusted" to "honest but curious... without any privacy constraints". Again, there is no new concept introduced, but well-known concepts are simply combined in order to provide the desired property.

The following sections will introduce the security and adversary model this approach is operating in. This way, a discussion about the targeted adversary and the trust and security assumptions is possible. Then the cryptographic and protocol prerequisites are provided in order to understand the implementation that follows. The paper concludes by providing some quantitative and qualitative analysis of the approach.

3.2. Security model

The main security parameter is denoted by $\lambda \in \mathbb{N}$. We write $a \leftarrow A(x)$ if a is assigned to the output of algorithm A with input x. An algorithm is efficient if it runs in probabilistic polynomial time (*ppt*) in the length of its input. For the remainder of this paper, all algorithms are ppt if not explicitly mentioned otherwise. If S is a set, we write $a \leftarrow_R S$ to denote that a is chosen uniformly at random from S. For a message $m = (m[1], m[2], \ldots, m[l])$, we call $m[i]$ a block, while $l \in \mathbb{N}$ denotes the number of blocks in a message m. For a list we require that we have unique, injective, and efficiently reversible encoding, which maps the list to $0, 1^*$.

The security model that covers the approach consists of three actors, namely the channel, the broker and the client. There are no security assumptions required for the channel except of eventual availability. The channel is allowed to be modelled as an unreliable broadcast channel. The memory of the broker can be modelled as public readable memory, thus no secrecy is required on the broker side concerning the management of the clientIDs. The client is required to be able to keep a secret which might be either completely intrinsic to the client e.g. provided by a Physical Unclonable Function (PUF), or injected via application level.

The security properties of the actors define the adversary model. The adversary which is assumed to be restricted to be in ppt only is allowed to have full knowledge about the channel and the memory of the broker at any time (*full-take*). The adversary, however, is not able to extract the secret from the client.

3.3. Prerequisites

In the following section the prerequisites are described in order to understand the solution. As this solution proposes a cryptographic approach within the given protocol specification, both aspects are described here.

3.3.1. Cryptographic prerequisites

The cryptographic protocol used for this solution is the Schnorr identification scheme as described in the RFC-8235. In order to render this paper self-contained, a brief summary of the RFC-8235 is given here. Please refer to the full RFC document for further details [11].

The Schnorr NIZKP can be implemented over a finite field or an elliptic curve. The technical specification is basically the same, except that the underlying cyclic group is different. For simplicity, this document describes the approach within the finite field, whereas in the RFC-document, both versions are described.

Let p and q be two large primes with $q|p-1$. Let $\mathbb{G}_q$ denote the subgroup of $\mathbb{Z}_p^*$ of prime order q, and g be a generator for the subgroup.

Statement $NIZKP[(x) : y = g^x]$ Alice knows the discrete logarithm x of the value y to the base g (within $\mathbb{Z}_p^*$).

Public input $p, q, g, \mathbb{Z}_p^*, \mathbb{G}_q$, and the cryptographic hash function $H(x)$.

Prover's (Alice) private input. $x \in \mathbb{Z}_q$ such that $y = g^x \mod p$.

P $\xrightarrow{(t,s)}$ **V** Alice chooses random $\omega \leftarrow_R \mathbb{Z}_q$, calculates $t = g^\omega$, calculates $c = H(g||t||y||\textit{public data})$, calculates $s = \omega + x \cdot c \mod q$

Verification. Bob *accepts* the proof only if $y \in \mathbb{Z}_p^*$ and $g^s = t \cdot y^c \mod p$.

3.3.2. Protocol prerequisites

The MQTT protocol [4] describes the possibility for authentication exchange in section 3.15 and section 4.12. The authors of the MQTT protocol were aware of the problem of session hijacking and replay attacks. Especially in section 5.4.5 of the protocol specification, they give hints on how to mitigate the attack attempts. But all these mitigation approaches are not focusing on the problem at hand but augment the security assumptions of the channel or even of the network, by requiring the channel to be secured (*TLS*) or even by asking for a secure network environment (*VPN*).

As already advertised in the security model, the approach does not require such heavy security assumptions. It is based on a challenge response mechanism already foreseen by the specification, very much like the described Salted Challenge Response Authentication Mechanism (*SCRAM*) in the non-normative part of the protocol description. Thus the approach perfectly fits within the given protocol boundaries.

3.4. Protocol description

The main idea of this protocol is to choose the *clientID* as a group element $y \in \mathbb{G}_q$. In the notation for the multiplicative group, the client chooses $y = g^x$ with $x \leftarrow_R \mathbb{Z}_q$. In order to void replay attacks, the broker provides the client with a nonce n to be used as *public data* for the Non-interactive Zero-Knowledge Proof. Only if the client is able to provide the proof (t, s) within the same network connection as the provided challenge, the broker proceeds with a CONNACK containing the reason code *0x00 (Success)*, in all other cases it disconnects with *DISCONNECT* and some reason code e.g. *0x83 (Implementation specific error)*. This way, the client provides an anonymous identification each time it establishes a fresh connection.

3.5. Implementation

A first approach would be a general purpose Schnorr NIZKP system, but the generality comes with a price, in the form of substantial processing time and large public parameters required to construct a proof [12].

The approach proposed in this paper consists in choosing a key pair (sk, pk) which verifies whether sk is the private key corresponding to the public key pk in a digital signature scheme. This choice allows us to reduce our proof of knowledge problem to that of digitally signing messages, whose implementation is simpler and more efficient than any known general purpose Schnorr NIZKP system [12].

Let the following be the primitives of a secure digital signature scheme:

$$Sign(sk, m) = s||m \tag{3.1}$$

$$Verify(pk, s||m) = \begin{cases} m, & \text{if s is valid} \\ \perp, & \text{otherwise} \end{cases} \tag{3.2}$$

Equation 3.1 signs the message m using the private key sk, and prepends the signature s to the message m. Equation 3.2 verifies whether s is a valid signature for m produced by the private key sk corresponding to the public key pk then outputs the original message m if the signature is valid, or $\perp$ if it is invalid [12].

It is assumed that the given digital signature scheme satisfies completeness, validity and zero-knowledge to be useful in this context.

The implementation requires some addition to the client and broker software, that provide the MQTT ability.

The following Figure 3.1 visualizes a the enhanced authentication process. For further examples within this paper the authentication method, which is needed to trigger enhanced authentication on the broker, is named *SMOKER*. The process uses the signature primitives mentioned in Equation 3.1 and 3.2 and the key pair $(sk, clientID \leftarrow_{mqtt} pk)$. As pk must serve as clientID, it must get mapped into the set of MQTT-*clientID*s. ClientIDs are further analyzed in Section 3.6.

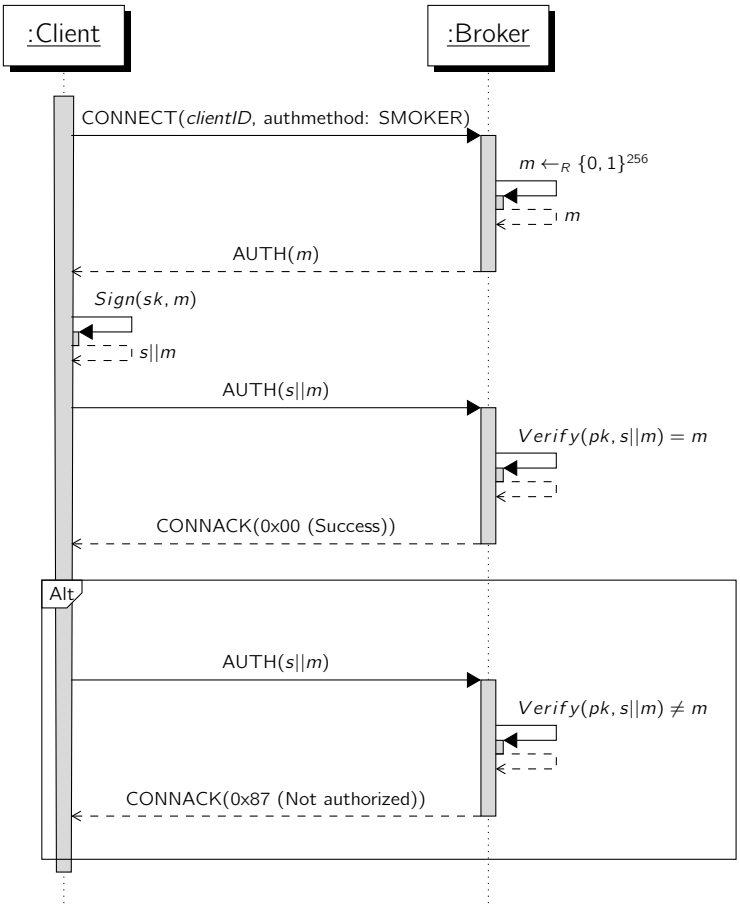


Figure 3.1.: Successful MQTT session establishment using the SMOKER authentication method.

Client Prior to the connection the client calculates its *clientID* by deriving a value pk from its secret key sk (can be the result of a puf or an out-of-band injected value). This is then mapped into the to the allowed set of MQTT-*clientID*s *clientID* $\leftarrow_{mqtt}$ pk. After the client established a connection to the broker, the client starts a MQTT session by sending the connect frame containing the authentication flag in expectancy of a broker authentication response, where a nonce $m \leftarrow_R \{0,1\}^{256}$ has to be provided. The client signs the received nonce using the secure signature scheme.

The client then sends the signed nonce $s||m$ as an authentication response within the same connection to the broker. If the client needs to re-establish the session, the very same procedure is executed as if the client would establish the session for the very first time.

Broker If a client is connecting to the broker, the provided *clientID* is only accepted, once the authentication succeeds within the same connection. The broker generates a random nonce $m \leftarrow_R \{0,1\}^{256}$ as a response to the first authentication frame within a new connection. The response delivered from the client consists of the signed nonce. The broker verifies the signature and thus extracts m' from $s||m$ which must be equal to the originally transmitted m. Only if this verification is successful, the *clientID* is accepted and the broker behaves in its usual way.

3.5.1. Connecting with an unknown authentication method

As shown in Figure 3.2, the given authentication method, requested by the client, must be supported by the broker – otherwise a CONNACK with the reason code 0x8C (Bad authentication method) [4, Table 3-1] must be returned and the connection is closed by the broker. An empty authentication method is not further handled by the extended authentication mechanism and is delegated to the brokers default implementation.

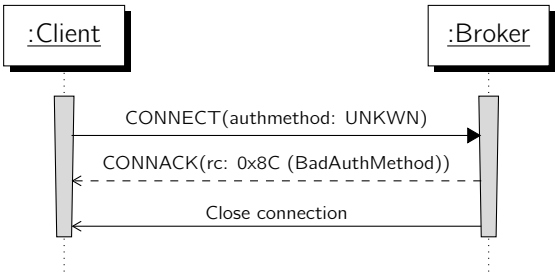


Figure 3.2.: MQTT connect with an unknown authentication method.

3.5.2. ClientID stealing

Common broker implementations are disconnecting an active client session if another client is connecting with the same clientID. The specification is very clear [4, section 3.1.3.1]:

> *"The broker must not accept connections with the same clientID"*

As the clientID does not need to be treated as a secret anymore, the broker must ensure that an unauthenticated client is not able to steal a clientID of a proper authenticated client. In this case the broker must reject the connection by sending a CONNACK packet with a *0x85 (Client Identifier not valid)* reason code. Conversely, a connecting client that successfully authenticates, must be preferred over any other unauthenticated client with the same clientID – the unauthenticated client must get disconnected immediately.

3.6. Analyses

First a fair warning: Due to the 'short' maximum length of the *clientID* covered by the MQTT specification in relation to cryptographic constraints, it is important to implement the proof or signature using the mathematics of elliptic curves. This way, the security parameter λ provided by the MQTT specification provides a maximum $\lambda = 70$. According to cryptographic sources [13], this is already a very low security. But using finite fields would lower λ by at least another factor of 20 and would not sustain an attack of any computational moderate potent adversary. Therefore, as an advice, the client and the broker must be enabled to accept a clientID set of order $\geq 2^{256}$. Using the alphabet supported by the MQTT specification this results in a minimum of 43 characters for the clientID in order to get a minimum $\lambda = 128$.

3.6.1. Energy and space requirements

In terms of computing power, the client must be able to execute two operations on an elliptic curve and one cryptographic hash per connection. The expensive calculations on the elliptic curve can even be lowered down to a single operation if the *clientID* derived from the according secret can be stored in the system. The computing power on the broker side must allow to calculate a minimum of two operations on the elliptic curve and one cryptographic hash per connection. Furthermore, the broker is required to provide a high entropy challenge nonce m per connection. Either, the broker gets true randomization, which is a very difficult task, or the broker uses an initialization vector (secret iv) seed to initialize a pseudo random generator function. It can keep seeding the pseudo random generation function by the answers it receives from the client connections. This way, the entropy of the client secret can be used to influence the available entropy on the broker, where natural sources of entropy are sparse by nature.

3.6.2. Passive adversary

The provided implementation allows the broker to reside in a very relaxed security model. No secret is required anymore concerning the *clientID*. The provided implementation further does not require any special treatment of the channel used for communication, as no secret is sent over the channel at all. As long as the underlying network protocol provides authenticity in terms of source and destination during connection, the MQTT session cannot be hijacked at all. The only remaining party within a strict security model is the client. It must be able to keep the secret safe from being extracted.

3.6.3. Active adversary

As long as the adversary is not in possession of the secret x kept by the client, it cannot provide a valid proof of knowledge to the broker. However, if the adversary model is relaxed, it can start a MITM attack. This allows the adversary to impersonate the client without the client nor the broker to gain any knowledge of that. Thus, the client is required to be sure about the public identity of the desired broker. This brings us back to TLS (and all its pros and cons) with server certificate, in order to provide server-identity 'only' in order to prevent the man-in-the-middle attack. So, one might argue, that the title of this paper might be miss leading, but it must be stressed out, that TLS is not required to prevent clientID stealing, but for broker identity in order to prevent the active adversaries MITM attack.

4. Client-Managed topic-based authorization

Authenticated parties are allowed to claim topics and authorize other clients to subscribe or publish on the claimed resource. Such a claim is fully client-managed – only the initiator can ever change or delete the claim again. If an authenticated client wants to subscribe or publish on a claimed topic, the broker must decide whether this is permitted or not on the basis of the available claims made by the clients.

4.1. Introduction

A MQTT client can basically do two things after it has established a connection to a broker: Publish or subscribe to a topic. Topics are the central resource of a client and (depending on a clients use case) worth protecting. Without protection, every client can publish or subscripe to a topic.

The MQTT 5.0 specification [4, Chapter 5.1] states:

> *"MQTT solutions are often deployed in hostile communication environments. In such cases, implementations will often need to provide mechanisms for: [...] Authorization of access to Server resources."*

There are various broker implementations (e.g. HiveMQ [1], Mosquitto [14]) that offer authorization options. However, these are typically managed by an admin on the broker side and can not be managed by the client itself. The approach presented in this thesis makes it possible for authenticated clients to independently manage the authorization rules of one or more topics. This behaviour allows clients to gain full sovereignty over specific topics on a broker. This will further be referred to as claiming.

A client can claim a topic by submitting a client-signed claim containing concrete topic-based permissions to the broker. The broker validates and stores this claim. Once a topic is claimed by a client, it becomes the owner of this claim and nobody but the owner can ever update it. Based on the claim store, managed by clients, the broker is able to decide whether access is permitted or not for usual publish and subscribe activities from clients.

4.2. State of the art

It is basically up to the broker implementation how the authorization is implemented. Common implemented mechanisms are the following [15]:

Access Control Lists ACL An ACL contains a list of authorizations rules. An authorization rule defines the access to a single resource and specifies which operations (e.g. publish, subscribe) may be performed per client. ACL's may be stored in a file or in a database on the broker – the persistence is handled differently per implementation.

Role Based Access Control (RBAC) RBAC is a role based authorization system. A role is an additional abstraction between the client and the resource to be protected. This procedure can be a simplification to maintain associations between technical rights and concrete clients. For example, existing role systems such as active directory or authorizations at operating system level could also be considered.

4.3. Security considerations

As mentioned in section Section 2.2 an attacker could manipulate the data source of the broker where claims are stored. Therefore, several precautions must be taken in order to ensure the attacker is not able to manipulate data to its favor.

Client-side signature of claims The broker must have the possibility to verify that a claim was not modified since creation. Therefore, the client must provide a signature of the given claim at creation time. The signature must be created with the secret counterpart of the clients public key that acts as clientID in the authenticated MQTT session. This allows the broker to verify a claim is valid when it evaluates it for further processing. If verification fails, the claim record must be considered compromised and may not further be used.

> **Brainstorming** That claims must be signed by the client was not clear for us from the beginning. The first idea came when we tested the creation of the first claims during development and were able to modify them very easily in a database tool. This finally led to the idea that a client must sign the provided claim.

Restricted area The restricted area is a dedicated branch within the tree of topics. The topic to be claimed, must start with a predefined segment followed by the clientID and the effective topic name (e.g. restricted/{clientID}/temperature). This implicitly creates a dedicated client space, which can only be managed by the clientID present in the topic name.

> **Brainstorming** For security reasons, we had to ensure that the deletion of claims by an attacker does not lead to unauthorized access. To prevent this, all topics that are not claimed should have been inaccessible to interrupt the data flow. This would make the broker unusable for unauthorized clients. We had to find a way to allow both access variants. This finally led to the idea of the restricted area.

Topic must contain the clientID Any user with access to the claim store could possibly modify an existing claim, and enter itself as the owner and use its self-calculated valid signature. The steal of the claim would allow it to listen to the communication on the topic without anyone ever noticing it. In order to prevent this, the clientID must be present in the topic name which has to be claimed. If this is not the case, the claim has to be considered to be invalid by the broker. This ensures that the relationship between signature, the owners' clientID and the topic name, is always upright.

> **Brainstorming** That claims must be signed by the client was not clear for us from the beginning. The first idea came when we tested the creation of the first claims while development and were able to modify them very easily in a database tool. This finally led to the idea of the client-side signature of the claims.

Claim flooding Any authenticated client is allowed to claim topics. A client can theoretically claim topics for itself at arbitrary intervals. Fortunately, MQTT does not allow to publish on wildcard topics [16, MQTT-3.3.2-2]. This prevents claiming wide topic ranges within one claim.

If the implementation of the persistence logic within the broker includes direct I/O access, it can lead to enormous loss of performance or even end up in a server crash. This problem is not further examined in this thesis. An approach to avoid flooding can be found in the thesis written by Swen Lanthemann [17].

4.4. Performance considerations

Storing claims The broker needs to store claims. It is a decision of the broker implementation whether this store is volatile or not. As claim/unclaim activities of clients are much rarer than usual publish/subscribe activity, the process to create, update or delete a claim should not be critical from the performance perspective.

Evaluating claims As MQTT is used within the field of IoT, there is potentially a lot of traffic the broker has to deal with. Therefore, intercepting every publish/subscribe activity to evaluate claims and the respective permissions could lead to a massive performance penalty. It is recommended to elaborate a clean solution design, which performs well for read activities to the claim store in order to avoid significant performance losses.

4.5. Implementation

The Figure 4.1 visualizes a successful sequence of a topic claim. This example is used for the subsequent definitions on client and broker side.

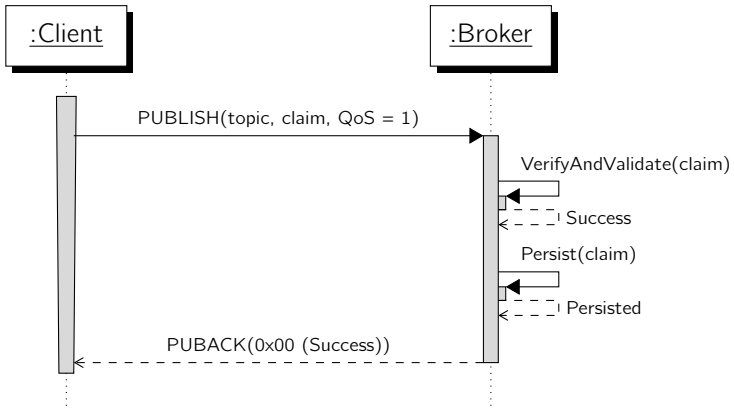


Figure 4.1.: Successful claim of a topic.

4.5.1. Client

Claim a topic The client initiates a topic claim by publishing a claim to a predefined system topic. This message must be published with QoS 1 as the broker submits feedback information within the PUBACK packet. An overview of the available QoS levels is given in Subsection 1.1.2.

The claiming payload sent by the client, is classified as a whitelist or blacklist. A restriction must contain at least one or more authorization rules consisting of the targeting topic and a set of rules for read (subscribe) and write (publish) actions. All containing rules are implicitly interpreted with the allow (whitelist) or deny (blacklist) operator based on the classification. Rules can be defined for all (wildcard) or specific clientID's.

If a client gets whitelisted on a topic, all other clients automatically get blacklisted and vice versa. The only exception is the owner itself, which is always guaranteed to have full access to all topics owned by it. In case a client only wants to authorize specific clients, it must know their clientIDs'.

Further, the client must sign the full restriction. As the client is fully responsible for his claims it needs to make sure that claims can not be manipulated by anyone else. Therefore, it's evident that the restriction which is provided to the broker must be signed with the clients private key. Thus, the broker is able to verify that the restriction it is looking up is not compromised.

The following reason codes are expected by the client after a claim has been published:

Reason code	Situation
0x00 (Success)	If the claim was successfully processed by the broker.
0x99 (Payload format invalid)	If the claim within the payload was invalid.

Table 4.1.: Expected reason codes of a claiming a topic.

Update claim Updates of claims are only accepted by the owner itself. An update can be triggered by publishing a new version of the claim on the same topic. The broker must then completely replace the previous claim with the new one.

The following reason codes are expected by the client after a claim update has been published:

Reason code	Situation
0x00 (Success)	If the update was successfully processed by the broker.
0x99 (Payload format invalid)	If the update claim within the payload was invalid.

Table 4.2.: Expected reason codes of a claim update.

Unclaim a topic If a client wants to unclaim a topic this must be achieved by publishing the concerned topic name within the payload on a specific unclaim topic. The Broker must then validate whether the client is allowed to delete it or not. If it is, the broker must delete this claim.

The following reason codes are expected by the client after an unclaim has been published:

Reason code	Situation
0x00 (Success)	If the unclaim was successfully processed by the broker.
0x99 (Payload format invalid)	If the topic to be unclaimed is actually not claimed on the broker.

Table 4.3.: Expected reason codes of an unclaim publish message.

Subscribing wildcard topics A client is basically allowed to subscribe any wildcard topic. Such subscriptions may also include resources where access to the subscribing client is not granted by the claim owner. However, the client may only receive messages it is allowed to.

The following reason codes in respect to the chosen QoS are expected after a client subscribes to a wildcard topic:

Reason code	Situation
0x00 (Granted QoS 0)	The subscription is accepted and the maximum QoS sent will be QoS 0.
0x01 (Granted QoS 1)	The subscription is accepted and the maximum QoS sent will be QoS 1.
0x02 (Granted QoS 2)	The subscription is accepted and any received QoS will be sent to this subscription.

Table 4.4.: Expected reason codes during a wildcard subscription respecting the QoS level.

Using claimed topics The following reason codes are to be expected by the client after subscribing or publishing to a topic within the restricted area:

Reason code	Situation
0x0{0, 1, 2} (Granted QoS {0, 1, 2})	If the subscription is authorized.
0x00 (Success)	If the publish message is authorized.
0x83 (Implementation specific error)	If the broker found claims for the requested topic that are classified to be compromised and therefore any traffic of this topic is rejected.
0x87 (Not authorized)	If the client is not authorized to access the topic.

Table 4.5.: Expected reason codes when subscribing within the restricted area.

4.5.2. Broker

Handling claims For each incoming claim configuration, the broker must perform various validations to complete the topic claim. If the validation fails, the broker returns a *0x99 (Payload format invalid)* reason code within the PUBACK packet. This can happen if at least one following cases occur:

- If the provided signature can not be verified successfully
- If the given topic contains MQTT specific wildcard characters (e.g. + or # [4])
- If the given topic is empty or contains only whitespaces

If the validation succeeds, the broker must store the claim and its corresponding signature. This store must be accessible throughout different sessions.

Handling publish and subscribe messages In addition to receiving topic claims, the broker must check for each incoming publish or subscribe activity whether the underlying client has permission to do so or not. This decision must be based on the persisted claims already submitted by other clients or the requesting client itself. A successful subscription is visualized in Figure 4.2 below:

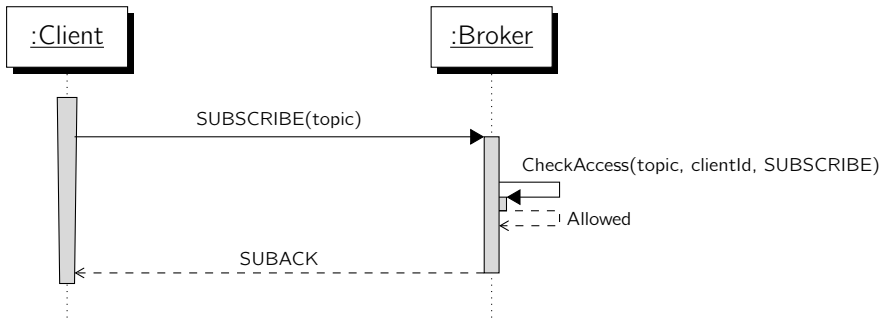


Figure 4.2.: Successful subscription of a claimed topic.

Before the broker can evaluate a specific claim, it needs to check its integrity with various validation rules. If one of the validation rules fails it must be assumed that the claim is compromised and can not be used for further evaluation. In this case the broker must return the 0x83 (Implementation specific error) reason code. The following rules must be validated:

- Verify the owners' signature to ensure the claim was not modified
- Check if the topic name contains the owners clientID

If the integrity check succeeds, the requested action will be allowed or denied based on the given claim permission. For allowed subscriptions, no further work is necessary on the broker side. An allowed publish messages gets enqueued on the broker to be delivered to authorized clients.

Because of the following reasons it may be mandatory to perform further security checks for any outbound message.

- The topic matches due a wildcard subscription of a client
- An existing topic subscription became invalid due to a claim update

The broker must accept wildcard subscriptions by clients without considering any permissions. If a publish message, matches a clients' wildcard subscription, the broker must evaluate whether the client is allowed to receive this message or not. If it's not, the message delivery must be discarded. Otherwise, the message is delivered as usual.

As claims are client-managed, they can be updated at any time. This may cause an authorized subscription, to become deprecated. It is a decision of the broker implementation whether permissions have to be enforced immediately or not – with the acceptance of the respective consequences.

Part II.

Implementation

5. Authentication implementation

This chapter will walk you through the steps which are required to get a working example of the concepts described in Chapter 3. The goal of this implementation is to provide a working PoC to illustrate the technical feasibility. The implementation provided is *not* meant to be a full production ready solution. Regardless, quality and stable code is important. Therefore, we try to automate testing where possible.

The implementation of the PoC is based on MQTTnet, a high performance .NET Core library for MQTT based communication. It is full cross platform compatible and provides a MQTT client and a server (broker). The library heavily relies on the builder pattern which provides a nice *fluent API* to set up the client and the broker.

5.1. Digital signature scheme

Every cryptographic operation is using the Edwards-curve Digital Signature Algorithm (EdDSA). It has been evaluated in Section 7.1.

5.2. ClientID

The clientID has to be converted to a Base32 encoded string containing the public key generated by EdDSA. Further information can be found in Section 7.2.

5.3. Field mappings

Before one can start implementing the protocol, the data which is used during the authentication process has to be mapped to the fields provided by MQTT. In Table 5.1 this mapping is defined. To avoid any errors, it's important that both parties participating in this protocol stick to the definitions provided below. Even though MQTT provides dedicated username and password fields which could be used to exchange any authentication data we don't rely on them because the whole point of this authentication scheme is to not submit any confidential information like for example a password.

MQTT	**Protocol Value**
AuthenticationMethod	SMOKER
AuthenticationData	Nonce generated by the broker
ClientID	Persistent public key

Table 5.1.: Field mappings.

Client Keypair Each client implementing this approach is responsible to create and store it's key pair. The key pair is a constant tuple, once generated or injected to the client it has to remain stable as long as this ClientID is in use. It's the responsibility of the client to keep the private key as safe as possible.

Brainstorming Our first approach was considering an initial and an ephemeral session key pair. We thought that implementing a session key pair, which is unique each time a new connection is established, improves the security by neglecting the possibility of replay attacks. During excessive discussion sessions, we started realizing, that our approach is already reply save due to the fact, that the nonce generated by the broker is already unique. The nonce consists of 256 randomly chosen bits which makes it practically impossible 2^{256} to hit the same nonce twice. A strong random number generator has to be used though.

5.4. Implementing the client

Implementing a client with MQTTnet is straight forward. Because the client library is already MQTT 5 capable, a method to provide an implementation of the *IMqttEnhancedAuthenticationExchangeHandler* interface is already present on the client-builder. A basic setup of the client is shown in Listing 5.1. The *ConfigManager* is a simple file writer which stores the generated key in the current users' profile path e.g. *$HOME*. Note that the *ConfigManager* is pretty basic as it stores the keys unencrypted. It just helps to keep the key pair persistent for repetitive use. However, the key pair can be injected by using the *MqttClientOptionsBuilder*. How the secret is stored on the system, is left to the client as stated in Chapter 2. On line 5, the *SmokerClientEnhancedAuthHandler* is responsible to encode the public key generated by the EdDSA signature scheme.

```
1  var config = new ConfigManager();
2  var keyPair = config.ReadKeypair();
3
4  var extendedAuthClientHandler = new SmokerClientEnhancedAuthHandler(keyPair);
5  var encodedClientId = extendedAuthClientHandler.EncodeClientId();
6  config.StoreKeypair(keyPair);
7
8  // Create TCP based options using the builder.
9  var options = new MqttClientOptionsBuilder()
10   .WithAuthentication(Smoker.AuthenticationMethod,
       extendedAuthClientHandler.InitialKeyPairA.PublicKey)
11   .WithClientId(encodedClientId)
12   .WithCommunicationTimeout(TimeSpan.FromMinutes(2))
13   .WithExtendedAuthenticationExchangeHandler(extendedAuthClientHandler)
14   .WithProtocolVersion(MqttProtocolVersion.V500)
15   .WithTcpServer("localhost")
16   .Build();
17
18 // Create a new MQTT client.
19 var client = new MqttFactory().CreateMqttClient();
20 await client.ConnectAsync(options);
```

Listing 5.1: Client configuration

The full source code of the enhanced authentication handler for the client can be seen in Listing 5.2.

```
1   public SmokerClientEnhancedAuthHandler()
2     : this(null)
3   {
4   }
5
6   public KeyPair KeyPair { get; }
7
8   public Task HandleRequestAsync(MqttExtendedAuthenticationExchangeContext
        context)
9   {
10    if (context.AuthenticationMethod != Smoker.AuthenticationMethod)
11    {
12      return Task.CompletedTask;
13    }
14
15    var nonce = context.AuthenticationData;
16    if (nonce == null || nonce.Length == 0)
17    {
18      return context.Client.DisconnectAsync();
19    }
20
21    var signedNonce = PublicKeyAuth.Sign(nonce, KeyPair.PrivateKey);
22
23    return context.Client.SendExtendedAuthenticationExchangeDataAsync(new
          MqttExtendedAuthenticationExchangeData
24    {
25      AuthenticationData = signedNonce,
26      ReasonCode = MqttAuthenticateReasonCode.ContinueAuthentication
27    });
28  }
29
30  public string EncodeClientId()
31  {
32    return Base32.Crockford.Encode(KeyPair.PublicKey, false);
33  }
```

Listing 5.2: Client Enhanced Authentication Handler.

5.5. Implementing a JavaScript based client

The protocol is designed with simplicity and portability in mind. This is important in order to reach as many clients as possible. Since libsodium is written in standard C, it is highly portable. This makes it possible to compile it on almost any platform and create appropriate wrappers. Therefore, a very simplified implementation for the widely used MQTT.js client library is provided in Listing 5.3. It uses the libsodium-wrappers package which can be installed via Node Package Manager (NPM).

```
(async () => {
    await ready;
    let initialKeypair = crypto_sign_keypair();
    const clientId =
        base32.encode(Buffer.from(initialKeypair.publicKey).toString());

    var client = connect('mqtt://127.0.0.1', {
        clientId: clientId,
        protocolVersion: 5,
        clean: true,
        properties: {
            authenticationMethod: 'SMOKER'
        }
    });
    client.on('packetreceive', function (p: Packet) {
        if (p.cmd.toString() === 'auth') {
            console.log(JSON.stringify(p));
            const packet = p as any;
            if (packet.properties.authenticationMethod == 'SMOKER') {
                const nonce = packet.properties.authenticationData;
                if (nonce) {
                    const signedNonce = crypto_sign(nonce,
                        initialKeypair.privateKey);
                    const authPackage = {
                        cmd: 'auth',
                        reasonCode: 24, // MQTT 5.0 code
                        properties: { // properties MQTT 5.0
                            authenticationMethod: 'SMOKER',
                            authenticationData: Buffer.from([
                                ...signedNonce
                            ]),
                            reasonString: 'continue',
                            userProperties: {
                                'test': 'test'
                            }
                        }
                    };
                    let clientEx = client as MqttClientEx
                    if (clientEx) {
                        client.connected = true;
                        clientEx._sendPacket(authPackage as Packet);
                        client.connected = false;
                    }
                }
            }
        }
    })
```

Listing 5.3: TypeScript implementation of the SMOKER authentication.

Unfortunately, it turns out that MQTT.js isn't fully MQTT 5.0 compliant and the enhanced authentication is missing. See Section 7.4 for further details.

The code seen on line 36-41 is just required to bypass some design flaws within MQTT.js library. In order to be able to send any packages to the broker, a connected state is expected. Internally the connected flag is only set, once a CONNACK packet is received, this obviously isn't necessary the case during an enhanced authentication, but we are still required to send packages.

5.6. Implementing the broker

If the broker fully supports the enhanced authentication, the implementation should be straight forward. Unfortunately MQTTnet didn't. See Section 7.3 for further details.

5.6.1. SMOKER Enhanced Authentication

Once the broker is capable of dealing with enhanced authentication methods, one can create an implementation of the *IMqttEnhancedAuthenticationBrokerHandler* interface. The implementation has to provide two methods, *StartChallenge* and *HandleAuth*. The StartChallenge method as it's name suggests is responsible for handling the CONNECT packet received from the client and initiate an enhanced authentication which returns an AUTH packet to the client.

As shown in Listing 5.4 on line 16, we create a nonce which is 32 bytes in size. Based on the underlying operating system the *GenerateRandomBytes()* method calls the appropriate function. For Windows, the *RtlGenRandom()* method and on recent Linux distributions the *getrandom()* system call is used. The public key is decoded from the CliendID and like the nonce stored for this authentication session. Therefore, both values can be used until the authentication protocol has finished or is aborted. Once prepared, we send the generated nonce back to the client with the correct reason code *0x18 (Continue Authentication)*.

As described in Section 5.4 the client signs the nonce and sends it back to the server. The server then calls the *HandleAuth()* method which verifies the signed nonce as shown in Listing 5.4 on line 36. If the signature is correctly verified, a sessionItem is stored. This indicates that the current session is successfully authenticated. Obviously only the broker is allowed to write session items. Afterwards a CONNACK packet is returned to tell the client that the authentication protocol has finished successfully.

```
1  public class SmokerEnhancedAuthBrokerHandler :
       IMqttEnhancedAuthenticationBrokerHandler
2  {
3    private byte[] _nonce;
4    private byte[] _publicKey;
5
6    public MqttBasePacket StartChallenge(MqttConnectPacket connectPacket)
7    {
8      var authMethod = connectPacket.Properties?.AuthenticationMethod;
9      if (string.IsNullOrWhiteSpace(authMethod))
10     {
11       // if no AuthenticationMethod is specified we continue with the default
12       return null;
13     }
14
15     if (!authMethod.Equals(Smoker.AuthenticationMethod))
16     {
17       return new MqttConnAckPacket
18       {
19         ReasonCode = MqttConnectReasonCode.BadAuthenticationMethod
20       };
21     }
22
23     _nonce = CryptoUtil.GenerateRandomBytes(32);
24     _publicKey = Base32.Crockford.Decode(connectPacket.ClientId).ToArray();
25     return new MqttAuthPacket
26     {
```

```
27          ReasonCode = MqttAuthenticateReasonCode.ContinueAuthentication,
28          Properties = new MqttAuthPacketProperties
29          {
30            AuthenticationMethod = Smoker.AuthenticationMethod,
31            AuthenticationData = _nonce,
32          }
33        };
34      }
35
36      public MqttBasePacket HandleAuth(MqttAuthPacket authPacketUpdate,
            IDictionary<object, object> sessionItems)
37      {
38        if (authPacketUpdate.Properties?.AuthenticationMethod !=
              Smoker.AuthenticationMethod)
39        {
40          return null;
41        }
42
43        var verificationResult = false;
44        try
45        {
46          var signedNonce =
                PublicKeyAuth.Verify(authPacketUpdate.Properties.AuthenticationData,
                _publicKey);
47          verificationResult = _nonce.SequenceEqual(signedNonce);
48        }
49        catch (CryptographicException)
50        {
51        }
52
53        var mqttReasonCode = verificationResult
54          ? MqttConnectReasonCode.Success
55          : MqttConnectReasonCode.ImplementationSpecificError;
56
57
58        if (verificationResult)
59        {
60          sessionItems.Add(CryptoConsts.AuthenticatedPair);
61        }
62
63        return new MqttConnAckPacket
64        {
65          ReasonCode = mqttReasonCode,
66          Properties = new MqttConnAckPacketProperties
67          {
68            AuthenticationMethod = Smoker.AuthenticationMethod
69          }
70        };
71      }
```

Listing 5.4: Enhanced Authentication Handler

5.6.2. ClientId stealing

To avoid the possibility of double ClientId's as introduced in Subsection 3.5.2, a connection validator is used. It basically favours a client which is able to prove its identity. In Listing 5.5 On line 20 an existing client with the same ID is searched. If one is found, and the according session is authenticated, the current connection is disconnected ungracefully with the reason code *0x85 (Client Identifier not valid)*.

```
1   public class SmokerConnectionValidator : IMqttServerConnectionValidator
2   {
3       private readonly IServiceProvider _serviceProvider;
4
5       public SmokerConnectionValidator(IServiceProvider serviceProvider)
6       {
7           _serviceProvider = serviceProvider;
8       }
9
10      public Task ValidateConnectionAsync(MqttConnectionValidatorContext context)
11      {
12          var mqttServer = _serviceProvider.GetService(typeof(IMqttServer)) as
                IMqttServer;
13          if (mqttServer == null)
14          {
15              return Task.FromResult(0);
16          }
17          return Task.Run(() =>
18          {
19              var statusList =
                    mqttServer.GetClientStatusAsync().GetAwaiter().GetResult();
20              var client = statusList.SingleOrDefault(s => s.ClientId ==
                    context.ClientId);
21              if (client != null &&
                    client.Session.Items.Contains(CryptoConsts.AuthenticatedPair))
22              {
23                  context.ReasonCode =
                        MqttConnectReasonCode.ClientIdentifierNotValid;
24              }
25          });
26      }
27  }
```

Listing 5.5: Validating a connection.

5.6.3. Mosquitto becomes a SMOKER

Even though it wasn't planed to make an implementation for Mosquitto, it just didn't feel right to completely omit it. Since libsodium is a plain C library, the crypto part is very similar compared to the MQTTnet implementation. The complete source code can be found within the gitlab mqttsec project. The implementation shown is not feature complete, only the authentication part is implemented.

The function definitions shown in Listing 5.6 are the most important hooks which need to be implemented in order to become fully SMOKER compatible.

```
1  /*
2   * Called after the plugin has been loaded. This will only ever be called
3   * once and can be used to initialize the plugin.
4  */
5  int mosquitto_auth_plugin_init(void **user_data, struct mosquitto_opt *opts,
6              int opt_count);
7
8  /*
9   * Only include this function in your plugin if you are making
10  * extended authentication checks.
11 */
12 int mosquitto_auth_start(void *user_data, struct mosquitto *client,
13             const char *method, bool reauth, const void *data_in,
14             uint16_t data_in_len, void **data_out, uint16_t *data_out_len);
15
16 /*
17  * Only include this function in your plugin if you are making
18  * extended authentication checks.
19 */
20 int mosquitto_auth_continue(void *user_data, struct mosquitto *client,
21             const char *method, const void *data_in,
22             uint16_t data_in_len, void **data_out,
23             uint16_t *data_out_len);
24
25 /*
26  * Called by the broker when topic access must be checked.
27  * extended authentication checks.
28 */
29 int mosquitto_auth_acl_check(void *user_data, int access, struct mosquitto *client,
30             const struct mosquitto_acl_msg *msg);
```

Listing 5.6: Mosquitto important plugin hooks

The implementation of those functions can be found in the smoker_auth_plugin.c file. The smoker_auth_plugin barely contains any logic though. It's just the entry point which wires everything together. The effective SMOKER code is defined in the *smoker.h* header as shown in Listing 5.7 and implemented in the *smoker.c* file.

```c
#define SMOKER_AUTH_METHOD_NAME "SMOKER"
#define SMOKER_PUBLIC_KEY_SIZE 32
#define SMOKER_NONCE_SIZE 32

enum smoker_reason_code {
  SMOKER_BAD_AUTHN_METHOD = 140,
  SMOKER_NOT_AUTHORIZED = 135 // 0x87 (135) - Not authorized
};

struct smoker_data
{
  unsigned char nonce[SMOKER_NONCE_SIZE];
  unsigned char public_key[SMOKER_PUBLIC_KEY_SIZE];
};

int smoker_copy(void **data_out, uint16_t *data_out_len,
                void *src, size_t nonce_len);
int smoker_init_data(void **user_data);
int smoker_set_clientid(struct smoker_data *data, const char *clientId);
int smoker_set_nonce(struct smoker_data *data);
int smoker_verify_nonce(struct smoker_data *data, const void *data_in,
                        uint16_t data_in_len);
```

Listing 5.7: smoker.h - SMOKER definitions.

The implementation of those functions is omitted, since the logic is basically the same as already described in Section 5.6. Once the plugin is compiled, one can enable it by adding the compiled shared module to the *mosquitto.conf* file. (e.g. *auth_plugin $(HOME)/projects/mosquitto-smoker/smoker_auth_plugin.so*).

6. Authorization implementation

This chapter will walk you through the steps which are required to get a working example of the concepts described in Chapter 4. The goal of this implementation is to provide a working PoC to illustrate the technical feasibility. The implementation provided is *not* meant to be a full production ready solution.

The authorization method described in this chapter, is only available for authenticated clients using the SMOKER authentication method described in Chapter 5. Nevertheless, unauthenticated users are still able to use the broker outside the restricted area. The restricted area is defined by a topic prefix, where topics only can be subscribed or published to, if the client is authenticated and having the required permission to do so.

6.1. Introduction

As MQTTnet offers the possibility to easily extend publish and subscribe functionality by implementing their interceptor interfaces, there was nearly no need to change the MQTTnet core code to implement the described authorization approach. The only change that was made to the MQTTnet code as part of the authorization, is extending the subscription interceptor context as mentioned in Subsection 7.5.1.

6.2. Restricted area

According to the Section 4.3, the broker must provide a restricted area, where clients can claim topics. If authenticated clients are granted, they can publish and/or subscribe to claimed topics within the restricted area. An example of a possible topic tree is shown in Figure 6.1:

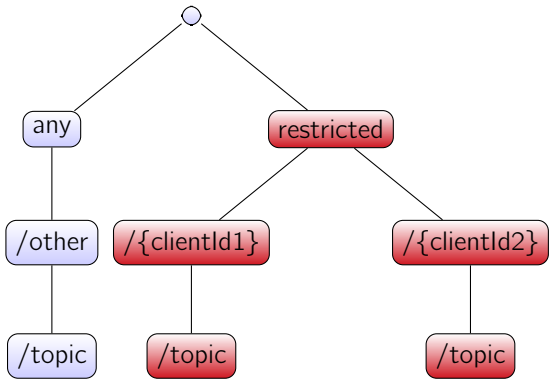


Figure 6.1.: Restricted topic area.

The nodes marked in red are located within this restricted area. The topic must be prefixed in the format *restricted/{clientId}/* where the *{clientId}* substitutes the claimers Base32 encoded clientID. The broker validates this topic format. If the validation fails the claim is rejected, all other topics outside the restricted area are treated as usual.

6.3. Reserved topics

The broker reserves topics to provide the claim and unclaim functionality. MQTT itself does not specify how administrative topics have to look like, however best practices exist and shall be used. It's common to use topics starting with the *$* character for such purposes [3]. According to that, the broker reserves the following topics:

Topic name	Purpose
$access/claim	The topic, where clients can publish claims to. Invalid claims published on this topic will be rejected.
$access/unclaim	The topic, where the client can release a claimed topic.

Table 6.1.: Field descriptions of the restriction class.

Reserve means, that clients can not subscribe to these topics.

6.4. Domain object model

The claim is the core domain object to build topic based authorization rules. Its structure is shown in Figure 6.2:

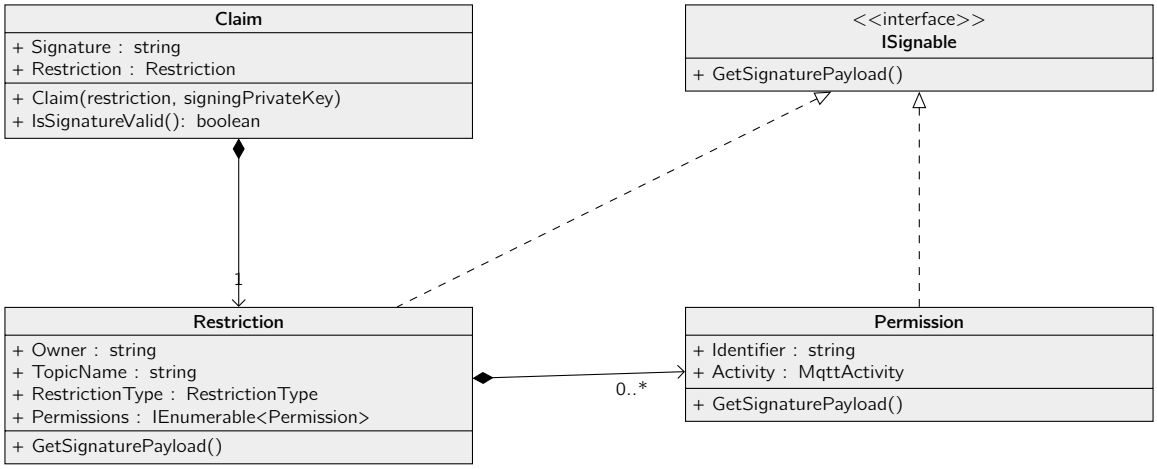


Figure 6.2.: UML diagram of a claim.

6.4.1. Claim

The claim itself only holds a Base64 string, representing the signature and a *Restriction* object. The *Restriction* is a required field, as a claim makes no sense without a Restriction and vice versa. The two objects are therefore related to each other through a composition as shown in Figure 6.2.

The signature of the *Restriction* is calculated within the constructor of the claim as shown in Listing 6.1:

```
1 public Claim(Restriction restriction, byte[] signingPrivateKey)
2 {
3     Restriction = restriction;
4     var bytes = PublicKeyAuth.Sign(restriction.GetSignaturePayload(),
          signingPrivateKey);
5     Signature = Convert.ToBase64String(bytes);
6 }
```

Listing 6.1: Constructor of the Claim object

The signature is generated with the clients private key which is the secret counterpart of the clientID, that is stored in the *Restriction::Owner* field. How these keys are generated is described in detail in Chapter 5. The signature is generated by the *Restriction::GetSignaturePayload* method which is implemented according to the *ISignable* interface. This function is described in detail within the *Restriction* description below. The signature is then stored as a Base64 string within the *Claim::Signature* field.

Once created, the claim can be verified by calling the *Claim::IsSignatureValid* method shown in Listing 6.2.

```
1 public bool IsSignatureValid()
2 {
3     byte[] signedHash = Convert.FromBase64String(Signature);
4     byte[] clientId = Base32.Crockford.Decode(Restriction.Owner).ToArray();
5     byte[] expectedResult = Restriction.GetSignaturePayload();
6
7     var verificationResult = PublicKeyAuth.Verify(signedHash, clientId);
8     return expectedResult.SequenceEqual(verificationResult);
9 }
```

Listing 6.2: Signature verification.

6.4.2. Restriction

The restriction merges a concrete topic with a set of permissions. It consists of an owner, a type, a topic name, and a list of permissions. The fields are described in Table 6.2 below.

Field	Description
Owner	The owner holds the clientID of the client claiming the topic. It is used by the broker, to verify the signature of the respective *Claim*. Since this value must be found within the topic name, it must not contain any wildcard (e.g. +, #) or delimiter (e.g. /) characters. Therefore the owner value must be restricted to a Base32 string. See Section 7.2 for more details.
TopicName	The topic name holds the MQTT topic which will be claimed. This is a standard MQTT topic which is always prefixed in the following format: *{restricted}/{clinetID}/*. This field is the key attribute for the broker to find and verify a claim. It is impossible that two claims share the same topic name within their restricted client space.
RestrictionType	This is an enumerated value which classifies the restriction to be a white- or blacklist. The type defines how the underlying *Permissions* are interpreted during permission evaluation.
Permissions	This list holds all permissions which results in an authorization pattern which is checked by the broker. This list is consulted if the topic gets accessed by other clients. The list may be empty which implies, that only the owner has permission on the given topic.

Table 6.2.: Field descriptions of the Restriction class.

The restriction implements the *ISignable* interface to provide the *GetSignaturePayload* function which is used to create the signature on the enclosing claim object.

```
1 public byte[] GetSignaturePayload()
2 {
3     var ownerBytes = Owner != null ? Encoding.UTF8.GetBytes(Owner) : new byte[0];
4     var topicNameBytes = TopicName != null ? Encoding.UTF8.GetBytes(TopicName) :
          new byte[0];
5     var restrictionTypeBytes = BitConverter.GetBytes((int)RestrictionType);
6
7     var permissionHashes = Permissions.Select(p =>
          p.GetSignaturePayload()).ToArray();
8     var byteArrays = new List<byte[]> {ownerBytes, topicNameBytes,
          restrictionTypeBytes};
```

```
9       byteArrays.AddRange(permissionHashes);
10
11      return CryptoUtil.Hash(byteArrays.ToArray());
12  }
```

Listing 6.3: Signature payload creation.

The method creates the according byte representations from the owner, topic and type fields. It then calculates the hashes of the underlying permissions. This list is passed to the *CryptoUtil* which concatenates all given byte arrays and creates a SHA-256 hash from it. To verify the resulting signature, the broker can call the exact same function to get the verification payload as shown in Listing 6.2

6.4.3. Permissions

The permission describes an access rule of the enclosing restriction. It actually holds only two fields described in Table 6.3 below:

Field	Description
Identifier	The identifier is targeting either to all or one specific clientID. To include all clients, the identifier string can be populated with the wildcard character (*). To target a specific client the corresponding clientID must be populated.
Activity	This is an enumerated value which defines the MQTT operation to protect. Allowed values are [*Publish* | *Subscribe*] | *All*].

Table 6.3.: Field descriptions of the Permission class.

If defining permissions, it is important to keep the *RestrictionType* in mind, because the permissions are negated if a blacklisting strategy is chosen compared to a whitelisting approach.

The permission also implements the *ISignable* interface because the restriction must include permissions into the signature payload as shown in Listing 6.3.

6.5. Interceptors and services

The following Figure 6.3 shows the interface and class structure implemented to process the claims. It shows the complete flow, from the arrival of the package at the interceptor until the storage of the respective claim in a data store.

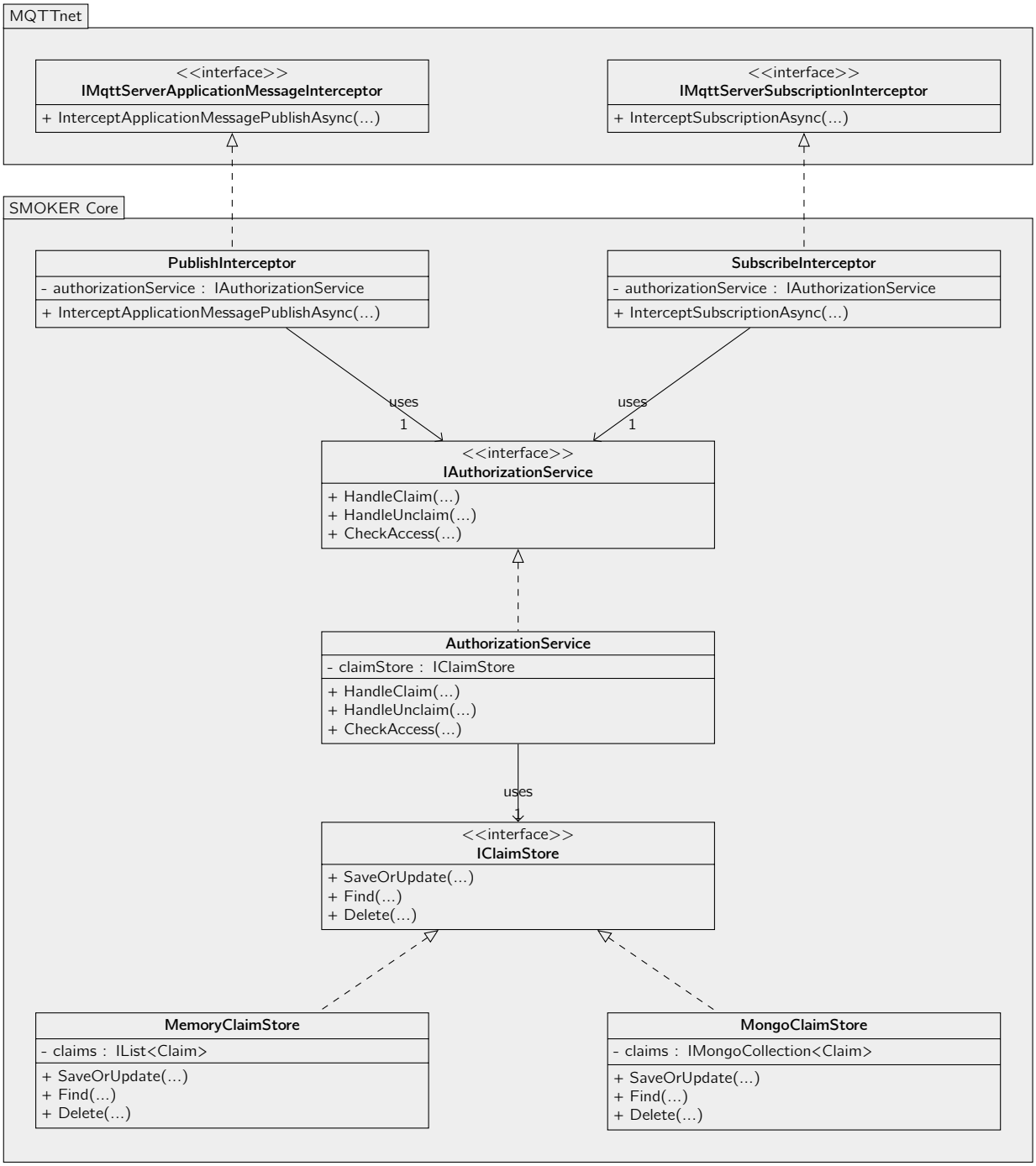


Figure 6.3.: UML diagram of interfaces and classes used for implement the authentication flow.

6.5.1. Publish interceptor

The *PublishInterceptor* class, is the implementation of the *IMqttServerApplicationMessageInterceptor* interface, where the function *InterceptApplicationMessagePublishAsync* gets triggered when ever a message gets published via the broker. The interceptor differentiates between three application messages:

- Creating or update a topic claim
- Unclaiming a topic
- Any other publish message

Create or update a claim Listing 6.4 shows the code to create/update claims that are published to the *$access/claim* topic. It is a defined system topic, created for this purpose only. The *PublishInterceptor* does only check whether the client is authenticated and deserializes the claim which results in a well defined JSON structure as defined in Listing 6.12. If deserialization succeeds, the interceptor passes the resulting claim object to the *IAuthorizationService* for further processing.

```
1  // Handle topic claim
2  if (context.ApplicationMessage.Topic.Equals(AuthorizationConsts.ClaimTopic))
3  {
4      // check if user is authenticated
5      if (!isAuthenticated)
6      {
7          throw new MqttCommunicationException("Client must be authenticated to
               claim topics");
8      }
9
10     // parse & handle claim
11     var claim = JsonConvert.DeserializeObject<Claim>(
12                 Encoding.UTF8.GetString(context.ApplicationMessage.Payload));
13     context.AcceptPublish = _authorizationService.HandleClaim(claim);
14 }
```

Listing 6.4: Publish interception to create or update a claim.

Unclaiming a topic Listing 6.5 shows the code to unclaim a topic. Messages to claim a topic are published to the *$access/unclaim* system topic. The payload of this message only contains the topic name to be unclaimed. Like claiming, the interceptor only works with authenticated clients. The payload is passed to the *IAuthorizationService* for further processing.

```
1  // Handle topic unclaim
2  if (context.ApplicationMessage.Topic.Equals(AuthorizationConsts.UnclaimTopic))
3  {
4      // check if user is authenticated
5      if (!isAuthenticated)
6      {
7          throw new MqttCommunicationException("Client must be authenticated to
               unclaim topics");
8      }
9
10     var owner = context.ClientId;
11     var topicName = Encoding.UTF8.GetString(context.ApplicationMessage.Payload);
12
13     context.AcceptPublish = _authorizationService.HandleUnclaim(owner, topicName);
14 }
```

Listing 6.5: Publish interception to unclaim a topic.

Any other publish message Other publish messages are passed to the *IAuthorizationService::CheckAccess* method where the service decides whether the client is allowed to publish or not.

```
// Handle all other publish actions
else
{
    context.AcceptPublish = _authorizationService.CheckAccess(context.ClientId,
        context.ApplicationMessage.Topic,
        MqttActivity.Publish, isAuthenticated);
}
```

Listing 6.6: Publish interception to check access.

6.5.2. Subscribe interceptor

The *SubscribeInterceptor* class, is the implementation of the *IMqttServerSubscriptionInterceptor* interface where the method *InterceptSubscriptionAsync* gets called whenever a client subscribes to a topic. The only thing the interceptor needs to do, is to check whether access for the requested subscription is given or not.

```
return Task.Run(() =>
{
    var isAuthenticated =
        context.SessionItems.TryGetValue(CryptoConsts.AuthenticatedPair.Key, out
        var isAuth) && (bool) isAuth;
    var accessAccepted = _authorizationService.CheckAccess(
        context.ClientId,
        context.TopicFilter.Topic,
        MqttActivity.Subscribe,
        isAuthenticated
    );
    context.AcceptSubscription = accessAccepted;
    context.ReasonCode = accessAccepted
        ? (MqttSubscribeReasonCode?) null
        : MqttSubscribeReasonCode.NotAuthorized;
});
```

Listing 6.7: Subscribe interception to check access.

6.5.3. Handle a claim

The *IAuthorizationService* is responsible to create, update or delete claims. A simplified version is given in Listing 6.8.

```
public bool HandleClaim(Claim claim)
{
    // Validate claim signature
    if (!claim.IsSignatureValid())
    {
        return false;
    }

    // validate restriction
    var restriction = claim.Restriction;
    var vc = new ValidationContext(restriction);
    IList<ValidationResult> results = new List<ValidationResult>();
    if (!Validator.TryValidateObject(restriction, vc, results, true))
    {
        return false;
    }

    // Persist
    var newRestriction = _claimStore.SaveOrUpdate(claim);
    var result = newRestriction != null;
    if (result)
    {
        Cache.AddOrUpdate(newRestriction.Restriction.TopicName, claim, (topicName,
            oldClaim) => claim);
    }
    return result;
}
```

Listing 6.8: Handle a claim request.

If a client creates or updates a claim, the request is received from the *PublishInterceptor*. The input is an already parsed claim, which can easily be used for further processing. In order to guarantee consistency, the claim has to pass several validation rules:

1. The claim signature is valid according to Section 6.4
2. The topic must not contain any wildcard characters (e.g. + and #)
3. The topic must be inside the restricted area – the first topic segment must be equal to *restricted*
4. The topic must be owned by the current client – the second topic segment must be equal to the owner string which is the clientID

Brainstorming The check if a topic is already claimed by another client may be obvious. Due to the security reason defined in Chapter 4, the clientID must be present within the topic name. Therefore, the mentioned validation is obsolete, as it is not possible to claim topics within other clients' space.

If the validation succeeds, the claim can be passed to the *IClaimStore::SaveOrUpdate* method which persist the claim. As soon the claim is successfully persisted, it is immediately active. Almost the same procedure is used to handle updates. The only difference worth to mention is within the *IClaimStore*, where the claim gets either created or replaced by the updated one. This behavior is also be known as an *upsert*.

To deal with unclaim requests, the *IAuthorizationService* provides the method *HandleUnclaim* which expects the owner (clientID) and the topic name as input. Those parameters are received from the *PublishInterceptor*. The service simply removes the record from the claim store and avoids any additional sanity checks.

```
public bool HandleUnclaim(string owner, string topicName)
{
    var deleteSuccess = _claimStore.Delete(owner, topicName);
    if (deleteSuccess)
    {
        Cache.TryRemove(topicName, out _);
    }

    return deleteSuccess;
}
```

Listing 6.9: Handle an unclaim request

6.5.4. Check access for publish and subscribe requests

Further, the *IAuthorizationService* has to check whether a client is allowed to process the intended MQTT activity or not. The *CheckAccess* method is implemented as follows:

```
public bool CheckAccess(string clientId, string topic, MqttActivity mqttActivity,
    bool isAuthenticated)
{
    // Dont allow subscriptions of reserved topics
    if (mqttActivity == MqttActivity.Subscribe &&
        TopicHelper.IsReservedTopic(topic))
    {
        return false;
    }

    // Topic is not in restricted area - access allowed for everyone
    if (!TopicHelper.IsTopicInRestrictedArea(topic))
    {
        return true;
    }

    // Topic seems to be in restricted area but client is not authenticated ->
        access denied
    if (!isAuthenticated)
    {
        return false;
    }

    if (!Cache.TryGetValue(topic, out var claim))
    {
        claim = _claimStore.Find(r => r.Restriction.TopicName ==
            topic).FirstOrDefault();
    }

    // topic not claimed -> access denied
    if (claim == null)
    {
        return false;
    }

    // claim signature cannot be verified -> compromised
    if (!claim.IsSignatureValid())
    {
        return false;
    }

    // topic is owned by caller -> implicitly allowed
```

```
39      if (claim.Restriction.Owner == clientId)
40      {
41          return true;
42      }
43
44      // Check if specific permissions are granted
45      var involvedPermissions = claim.Restriction.Permissions.Where(p =>
            p.Identifier == clientId || p.Identifier ==
            AuthorizationConsts.AnyClientIdentifier);
46      var permissionMatch = involvedPermissions.Any(p => p.Activity == mqttActivity
            || p.Activity == MqttActivity.All);
47
48      switch (claim.Restriction.RestrictionType)
49      {
50          case RestrictionType.Whitelist:
51              return permissionMatch;
52          case RestrictionType.Blacklist:
53              return !permissionMatch;
54          default:
55              return false;
56      }
57  }
```

Listing 6.10: Check access of publish ans subscribe requests

As seen in Listing 6.10, several checks are made to allow or deny access to the requested resource. The mentioned checks in detail:

1. If the topic is not within the restricted area, by not having the first segment of the topic equal to "*restricted*", the access is granted.
2. If the topic is within the restricted area, but the user is not authenticated, the access is denied.
3. If the topic is within the restricted area, but is not yet claimed by any client, the access is denied.
4. If the signature of the matching claim can not be verified, the claim seems to be compromised and thus the access is denied.
5. If the matching claim is owned by the current client, the access is allowed.
6. Whether the restriction and the according permissions are resulting either granted or denied.

By default, access to topics within the restricted area is denied unless the rules described above are resulting in an allowed state.

Brainstorming For the sake of clarity, log statements within the code examples were removed. However, it should be noted that log statements have a huge performance impact, in case the broker is under heavy data load or gets stress tested. Therefore, the log level was set to *Error* by default. The *Debug* level must not be used for production or benchmarking.

6.6. Data store

Implementations of the *IClaimStore* interfaces can be used as a data store for claims. The claim store needs to provide basic functions in order to find, create, update and delete claims. Currently, there are two implementations of this interface available.

6.6.1. In-Memory

The *MemoryClaimStore* contains a list of claims in-memory and does not persist them in any way. If the broker gets restarted or the power is interrupted, the data of this list is irretrievably lost. Therefore, this implementation is not suitable for a production environment. It is very handy for testing purposes, as there is no need to set up another environment like a database.

6.6.2. Mongo database

The *MongoClaimStore* stores the claims within a Mongo database. This implementation requires a running Mongo database environment to which the broker can connect to. The connection can be configured in the hosting environment's configuration. An example of such a configuration can be found in Subsection 11.1.1.

6.7. Claim submission

The submission of claims is carried out exclusively by the clients. To make use of the new broker authorization functionalities, the MQTTnet client theoretically does not need to be extended. It only has to ensure that the claims are transmitted to the broker in the correct format and on the correct topic. Nevertheless, some helper functions have been implemented to simplify the use of the authorization ability.

6.7.1. Client extensions

To improve the usability, two extension methods have been implemented on the *IMqttClient* interface. This interface is provided by the MQTTnet library. The implementations are shown in Listing 6.11.

```
1  public static class MqttClientExtensions
2  {
3      public static async Task<MqttClientPublishResult> ClaimAsync(this IMqttClient
           client, Claim claim)
4      {
5          var msgClaim = new MqttApplicationMessage
6          {
7              Payload = Encoding.UTF8.GetBytes(JsonConvert.SerializeObject(claim)),
8              Topic = AuthorizationConsts.ClaimTopic,
9              QualityOfServiceLevel = MqttQualityOfServiceLevel.AtLeastOnce
10         };
11
12         return await client.PublishAsync(msgClaim);
13     }
14
15     public static async Task<MqttClientPublishResult> UnclaimAsync(this
           IMqttClient client, string topic)
16     {
17         var unclaimMsg = new MqttApplicationMessage
18         {
19             Payload = Encoding.UTF8.GetBytes(topic),
20             Topic = AuthorizationConsts.UnclaimTopic,
21             QualityOfServiceLevel = MqttQualityOfServiceLevel.AtLeastOnce
22         };
23
24         return await client.PublishAsync(unclaimMsg);
25     }
26 }
```

Listing 6.11: *IMqttClient* extension methods

The two functions are actually wrapping a default *IMqttClient::PublishAsync* call, but assist to prepare the application message according to the brokers' expectations. Note that the *ClaimAsync* method must also be used to update an existing claim.

6.7.2. Serialization

As seen in Listing 6.11 on line 7, the claim gets serialized into a JSON representation. More about serialization and its performance can be read in Section 7.6. For claiming activity, the broker expects a JSON payload that complies with the pseudo schema shown in Listing 6.12.

```
{
    "Signature": < Base64 encoded signature of the Restriction property >,
    "Restriction": < The restriction object which was signed >
    {
        "Owner": < Base32 encoded public key >,
        "TopicName":  < Topic in format 'restricted/{Owner}/any/topic' >,
        "RestrictionType": < 'Blacklist' or 'Whitelist' >,
        "Permissions": < A list of permissions >
        [
            {
                "Identifier ": < '*' or a Base32 encoded clientID >,
                "Activity": < 'Publish', 'Subscribe' or 'All' >
            }
        ]
    }
}
```

Listing 6.12: Pseudo JSON schema for claims.

An example of a valid claim serialized in JSON, is shown in Listing 6.13:

```
{
    "Signature": "z1RZr4XffEB{ ... }/l+0uEylJ3omN",
    "Restriction": {
        "Owner": "9X8VA8YD{ ... }5X73XY40",
        "TopicName": "restricted/9X8VA8YD{ ... }5X73XY40/chat/private",
        "RestrictionType": "Blacklist",
        "Permissions": [
            {
                "Identifier": "*",
                "Activity": "Subscribe"
            }
        ]
    }
}
```

Listing 6.13: Example of a JSON serialized claim.

The client publishes such payloads on the claim topic. These messages will be received by the broker and intercepted by the *PublishInterceptor* as described in Subsection 6.5.1.

Part III.

Methodology

7. Solution description

This chapter will give you background information about the thinking process and decisions made in order to get the implementation done. It's a dedicated chapter because it's not necessary to fully understand every detail during implementation but it is provided for the sake of completeness.

7.1. Choosing a digital signature scheme

According to Section 3.5, a digital signature scheme has to be chosen. The scheme has to take into account that IoT devices, which can be heavily resource constrained when it comes to bandwidth, computing power and/or energy consumption, might be used. Therefore, it's important that the cryptographic algorithms used, are as efficient as possible. See Subsection 3.6.1 for details.

Event though elliptic curves are probably the best choice, a short overview of the most commonly used signatures schemes is provided.

7.1.1. RSA Signature Scheme (1978)

The RSA signature scheme is based on RSA encryption. It's security relies on the difficulty of factoring a product of two large primes. The RSA signature scheme has emerged as the most widely used digital signatures scheme in practice [6, Chapter 10.2].

Computational Aspect The RSA signature size is dependent on the size of the modulus. The size is typically in the range from 1024 to 3072 bits. Therefore, our signature has a length between 128-384 bytes. Even though such a signature length is not a problem for most internet applications, it can be undesirable in systems which are constrained like IoT or mobile devices [6, Chapter 10.2.2].

7.1.2. Elgamal Digital Signature Scheme (1985)

The Elgamal signature scheme is based on the difficulty of computing discrete logarithms. Unlike RSA, where encryption and digital signature are almost identical operations, the Elgamal digital signature is quite different from the encryption scheme with the same name.

Computational Aspect Because the security of the signature scheme relies on the discrete logarithm problem, we need to choose a large prime p which should be at least 1024 bits in length. The signature consists of the pair (r, s). Both have roughly the same bit length as p, so that the total length of the package $(x, (r, s))$ is about three time as long as only the message x [6, Chapter 10.3.2].

7.1.3. Digital Signature Algorithm – DSA (1991)

The Elgamal signature algorithm is rarely used in practice. Instead, a much more popular variant is used, known as the *Digital Signature Algorithm (DSA)*. It is a federal US government standard for digital signatures and was proposed by the National Institute of Standards and Technology (NIST) [6, Chapter 10.4].

Computational Aspect It's main advantage over the Elgamal signature scheme is that the signature is only 320 bits long. The main idea of DSA is that there are two cyclic groups involved. One is the large cyclic group $\mathbb{Z}_p^*$ with an order of 1024 bits. The second one is in the 160 bits subgroup of $\mathbb{Z}_p^*$. This setup allows shorter signatures. Like the Elgamal signature scheme, the DSA signature consists of a pair of integers (r, s). Since both of the parameters are only 160 bit long, the total signature length is 320 bits. The Table 7.1 gives an overview of the most commonly used bit lengths.

p	q	Signature
1024	160	320
2048	224	448
3072	256	512

Table 7.1.: Bit lenght of common parameters for DSA.

7.1.4. Elliptic Curve Digital Signature Algorithm –ECDSA (1992)

Elliptic curves have several advantages over RSA and schemes like Elgamal or DSA. In particular, in absence of strong attacks against Elliptic Curve Cryptography (ECC), bit lengths can be significantly shorter. An overview is given in Table 7.2.

Algorithm Family	Cryptosystems	Key Length (bit)			
Integer factorization	RSA	1024	3072	7690	15360
Discrete logarithm	DH, DSA, Elgamal	1024	3072	7690	15360
Elliptic curves	ECDH, ECDSA, EdDSA	160	256	384	512

Table 7.2.: Bit lengths of public-key algorithms [6, Table 6.1].

Computational Aspect: The shorter bit length of ECC often results in shorter processing time and in shorter signatures.

7.1.5. Edwards-curve Digital Signature Algorithm – EdDSA (2011)

ECDSA and EdDSA are not related to each other, except that the mathematics behind, involves elliptic curves. The EdDSA is based on the Schnorr signature algorithm and relies on the difficulty of the Elliptic Curve Discrete Logarithm Problem (ECDLP) problem. Compared to ECDSA, the signature creation is deterministic and therefore it's not possible to foolproof session keys. EdDSA works over twisted Edwards curves [18], which have a few nice properties. The most used twisted Edwards curve is probably the Montgomery curve Curve25519. Bernstein and his team are the creator of the Curve25519, which is the foundation of the EdDSA signature scheme which uses the methods provided in Ed25519.

Computational Aspect: The Ed25519 is especially interesting because it features the following characteristics [19]:

- *Fast single-signature verification*
- *Very fast signing*
- *Fast key generation.* Key generation is almost as fast as signing. There is a slight penalty for key generation to obtain a secure random number from the operating system.
- *High security level.* This system has a 2^{128} security target; breaking it has similar difficulty as RSA with 3000-bit keys.

- *Foolproof session keys.* Signatures are generated deterministically; key generation consumes new randomness but new signatures do not. This is not only a speed feature but also a security feature, directly relevant to the recent collapse of the Sony PlayStation 3 security system [20].
- *Collision resilience.* Hash-function collisions do not break this system. This adds a layer of defense against the possibility of weakness in the selected hash function.
- *Small signatures.* Signatures fit into 64 bytes. These signatures are actually compressed versions of longer signatures; the times for compression and decompression are included in the cycle counts reported above.
- *Small keys.* Public keys consume only 32 bytes. The times for compression and decompression are again included.

7.1.6. Evaluation

According to the properties mentioned above, the EdDSA seems to perfectly suit our needs considering the restrictions mentioned in Subsection 3.6.1. Even tough the mathematics introduced in RFC-8235 [11] doesn't seem to be that complicated at first. Dealing with large numbers, which are required to implement the ECDLP, can quickly become confusing. Folks who work in the space of applied cryptography usually follow the mantra: *Don't implement your own crypto*. Considering this mantra, the approach showing, is using the library libsodium respectively Sodium.Core, which provides C# proxy classes to wrap the C interop calls made to libsodium. libsodium is based on NaCl (pronounced "salt"), a library written by Daniel J. Bernstein, which is one of the leading experts in the space of cryptography especially when it comes to elliptic curves.

7.2. ClientID considerations

The MQTT specification limits the ClientID length to 23 characters [4, MQTT-3.1.3-5]. The most commonly used brokers allow much longer ClientID's, in fact they are barely limited.

The MQTT specification dictates [4, Chapter 3.1.3.1]:

> *"The Server MAY allow ClientID's that contain more than 23 encoded bytes. The Server MAY allow ClientID's that contain characters not included in the list given above."*

Our approach uses a public key generated by the EdDSA, which is 256bit in length. We need to encode those bytes in order to get an ASCII representation of the public key. We choose the lesser known Base32 as binary-to-text encoder, because base64 uses [/, +] as valid characters for encoding. Those two characters are invalid if you stick to the specification. Furthermore, the ClientID is used within a topic and the authorization approach introduced in this thesis does not allow to contain those characters. Base32 is able to encode 8 bits to 5-bit Base32. Therefore, a minimum character length of $\frac{256}{5} \approx 52$ is required for the ClientID. Because of how Base32 encodes the bytes, additional characters may be included for padding, which deals with any rounding issues. It is possible to disable this padding behaviour, which can lead to a different ClientID length. Since we don't really care about +/- one character, we did turn off the padding. We end up with a final ClientID length of **53** characters. An example of a calculated ClientID can be seen in Listing 7.1.

```
FG11KZANSHWZNR77BRWTGA57TVDF7ZRX007RXDD6YBKXQH41HHGG
```

Listing 7.1: ClientID example

Therefore, our CliendID consists of an arbitrary string randomly chosen from 32^{53}. Which makes ClientId collisions very unlikely.

7.3. MQTTnet and enhanced authentication

Even though the authors of MQTTnet [16] maintain to this day that they are MQTT 5.0 compliant, it turns out that they are completely lacking the new enhanced authentication feature on the broker side. On github.com [21] one can find a corresponding open issue [22].

Since we are engineers, we've decided to extend the existing broker and implement the missing parts by ourselves. It was definitely a challenging journey. But in return, we got even more familiar with the MQTT protocol and learned a lot about async programming, especially considering Transmission Control Protocol (TCP) sockets. For us, it's a matter of course, that we make the created code available to the great open source community. A corresponding pull request has been created (#835) [23].

As shown in Figure 7.1 we just extend the existing *MqttServerOptionsBuilder* with the method *WithEnhancedAuthenticationExchangeHandler* which takes an *IMqttEnhancedServerAuthenticationExchangeHandler* as parameter. This signature perfectly suits the one provided by the client and should therefore match the existing architecture given by MQTTNet.

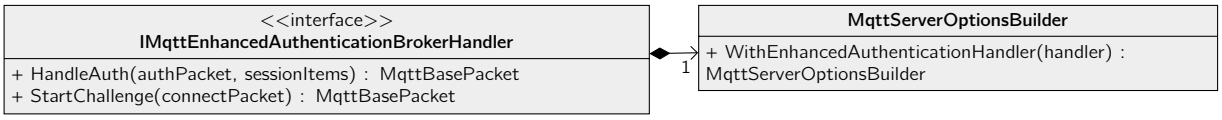


Figure 7.1.: MQTTnet enhanced authentication extensibility.

After the easy part, we had to extend the current functionality to be able to perform the enhanced authentication challenges. Those challenges have to continue until the broker sends a CONNACK. For the sake of completeness these are the most relevant parts of the specification [4] to consider:

- If the server requires additional information to complete the authentication, it can send an AUTH packet to the client. This packet MUST contain a Reason Code of 0x18 (Continue authentication). If the authentication method requires the server to send authentication data to the client, it is sent in the Authentication Data.
- The client responds to an AUTH packet from the server by sending a further AUTH packet. This packet MUST contain a reason code of 0x18 (Continue authentication). If the authentication method requires the Client to send authentication data for the server, it is sent in the Authentication Data.
- The client and server exchange AUTH packets as needed until the server accepts the authentication by sending a CONNACK with a reason code of 0. If the acceptance of the authentication requires data to be sent to the client, it is sent in the Authentication Data.

To do so, the existing class *MqttClientConnection* had to be extended to allow the custom handler shown in Figure 7.1, to do it's work.

As shown in Listing 7.2 we first check if an *EnhancedAuthenticationBrokerHandler* is configured. If this is the case we start the authentication process by calling the *StartChallenge* method and send the packet created back to the client. If no packet was created e.g. no handler is configured, the default implementation of a CONNACK packet is returned (Line: 6-21).

Then the broker internally starts listening for further packets which the client might send. If the packet received is a *MqttAuthPacket*, the packet is forwarded to the *HandleAuth* method, which internally handles the packet accordingly (Line: 26-41).

```
1  private async Task<MqttClientDisconnectType> RunInternalAsync()
2  {
3    ...
4    // previous code in this method has truncated
5
6    var authPacket = _serverOptions.EnhancedAuthenticationBrokerHandler?
7                       .StartChallenge(ConnectPacket);
8    if (authPacket != null)
9    {
10       await SendAsync(authPacket).ConfigureAwait(false);
11   }
12   else
13   {
14       await SendAsync(
15         new MqttConnAckPacket
16         {
17             ReturnCode = MqttConnectReturnCode.ConnectionAccepted,
18             ReasonCode = MqttConnectReasonCode.Success,
19             IsSessionPresent = !Session.IsCleanSession
20         }).ConfigureAwait(false);
21   }
22
23   ...
24   // Code skiped
25
26   if (packet is MqttAuthPacket clientAuthPacket)
27   {
28       if (_serverOptions.EnhancedAuthenticationBrokerHandler == null)
29       {
30           continue;
31       }
32
33       var responsePacket = _serverOptions.EnhancedAuthenticationBrokerHandler
34                                .HandleAuth(clientAuthPacket, Session.Items);
35       if (authPacket is MqttConnAckPacket connAckPacket)
36       {
37           connAckPacket.IsSessionPresent = !Session.IsCleanSession;
38       }
39       await SendAsync(responsePacket).ConfigureAwait(false);
40       continue;
41   }
42
43   ...
44   // Code skiped
45 }
```

Listing 7.2: Client configuration

That's basically all we had to implement to make MQTTnet ready to deal with an enhanced authentication. Even though the code produced, only takes up a couple of lines, it took it's time until we understood the existing source and came up with this simple solution.

7.4. MQTT.js and enhanced authentication

It turns out that MQTT.js only partially supports MQTT 5.0. Of course the part which is required to handle the enhanced authentication is missing, and we're forced to write our own implementation. A branch with a basic

implementation can be found on our fork of MQTT.js [24]. An official pull request hasn't created yet, since this is an alpha version used for demonstration purposes only. Our intention is to create an official pull request (probably after the thesis) and push MQTT to 5.0.

7.5. MQTTnet and authorization

The approach introduced in this thesis implements the MQTTnet publish and subscribe interceptor interfaces to extend the broker with the new authorization features. It is important to understand when exactly MQTTnet triggers interceptors in order to implement these features.

7.5.1. Subscribe interceptor

The subscription interceptor provided by MQTTnet is triggered before the brokers SUBACK response. Therefore, it is possible to return a feedback in form of MQTT reason codes within the brokers SUBACK packet. However, MQTTnet does not provide a possibility to set a custom reason code for a rejected subscription yet. By default, MQTTnet returns the 0x80 (Unspecified error) reason code if a subscription is rejected.

The interceptor implemented in this thesis is actually only checking for authorization. In a negative outcome of this check, the 0x80 reason code would be misleading as there is a reason code 0x87 (Not authorized) [4, Table 3-8] specified by the MQTT specification. Therefore, we decided to extend the interceptor context class from MQTTnet to provide the possibility to set a custom reason code which is added to the SUBACK response.

7.5.2. Publish interceptor

The publish interceptor provided by MQTTnet is an outbound interceptor. This means, it is triggered when the broker dequeues already received messages to deliver the messages to clients. In comparison, the inbound interceptor would be triggered before the message is enqueued and before the PUBACK packet is responded to the client.

As MQTTnet only provides an outbound interceptor, it is currently not possible to return feedback in form of reason codes to the client. The client will therefore always receive the 0x00 (Success) reason code for publishes to claim a topic. Actually MQTT specifies reason codes such as 0x99 (Payload format invalid) which would be predestined for the use case implemented in this thesis. But since we are implementing a PoC, we accepted the circumstance that reason codes for publish interceptions can not be customized. The core functionality can also be implemented with an outbound interceptor – albeit slightly less elegant than with an inbound interceptor.

7.6. Serialization

As a client needs to transmit data (claims) to the broker, a decision on how to serialize the payloads had to be made. These days, the first idea is of course JSON. Almost every programming language can easily handle it as it is very common and human readable. However, there are a lot of possibilities to serialize data together with JSON. As we are operating within the IoT field, it could be worth investigating different opportunities. As we implemented a PoC, we explicitly wanted a human readable format for the sake of simplicity. Therefore, we decided to use the JSON format anyway. But when it comes to refining the performance of serialization, other approaches like binary formats should be studied. For example Google's protobuf may produce up to 34% smaller payloads with 21% less computation time compared to JSON within a JavaScript environment [25].

8. Broker evaluation

This chapter describes the evaluation process of the MQTT broker which will be used to implement the SMOKER extension. The focus will be in particular on extensibility of authentication and authorization parts of the broker. Furthermore, the selected broker code-base should be as simple as possible as the implementation part of this thesis is targeting a PoC and not a full production ready solution.

8.1. Broker and client evaluation

In order to be able to realize the enhanced authentication, an already implemented MQTT broker has to be evaluated. There are numerous implementations available which makes it quite challenging to choose a suitable one. To bring down the number of possible brokers a few requirements have to be defined.

8.2. Requirements

In order to work efficiently we enforce some requirements which our MQTT broker needs to fulfill.

#	Requirement	Description
1	MQTT v5	Since the enhanced authentication procedures were specified with MQTT 5, the broker ideally already implements the latest version of the protocol. MQTT 5 was only released in March 2019 – so the number of brokers which implement the latest version of the protocol is still modest.
2	Open source	When implementing advanced authentication procedures, core components may need to be adapted. In addition, insight into the source code often helps to understand key concepts of the application design and it's inner semantics.
3	Programming language	To keep the implementation as efficient as possible, the broker should be written in a programming language already known to the project team. One of the following is required: • Java • C# • JavaScript
4	Extensibility	Many broker implementations provide extension possibilities to prevent modifications of core functionality. A well designed extension mechanism can facilitate the implementation of enhanced authentication.
5	Documentation	A detailed documentation is desirable, therefore the implementation can be made without much code inspection, debugging and reverse engineering.

Table 8.1.: Broker requirements.

8.3. Brokers

The following brokers were chosen for evaluation and checked against the requirements found in Table 8.1. Since the number of MQTT 5 brokers is still very limited, brokers that only support version 3.1.1 of the protocol have been added to the list. It would also be conceivable to develop the most necessary features for the enhanced authentication by our self.

Name	Language	Version	Docs	License	Extensible API
Mosquitto	C	5.0	OK	EPL/EDL	Yes
HiveMQ	Java	5.0	OK	Apache 2	Yes
Moquette	Java	3.1.1	Spare	Apache 2	No
VerneMQ	Erlang	5.0	OK	EPL/EDL	Yes
MQTTnet	C#	5.0	OK	MIT	Yes
Vert X	Java	3.1.1	OK	Apache 2	No

Table 8.2.: Possible brokers.

Values marked in red, fail to comply with one of the requirements. Only two brokers, which are marked green, are left namely HiveMQ and MQTTnet. We will have a closer look at those and try to make an educated guess on which broker to choose. Especially the already implemented authentication and authorization methods of those brokers are examined.

8.4. Broker security

In general, security in MQTT is divided in three levels.

8.4.1. Network level

One way to provide a secure and trustworthy connection is to use a physically secure network or VPN for all communication between clients and brokers. This solution is suitable for gateway applications where the gateway is connected to devices on the one hand and with the broker over VPN on the other side.

8.4.2. Transport level

When confidentiality is the primary goal, TLS is commonly used for transport encryption. This method is a secure and proven way to make sure that data can't be read during transmission and provides client-certificate authentication to verify the identity of both sides.

8.4.3. Application level

On the transport level, communication is encrypted and identities are authenticated. The MQTT protocol provides a clientID and username/password credentials to authenticate devices on the application level. These properties are provided by the protocol itself. Authorization or control of what each device is allowed to do, is defined by the specific broker implementation. Additionally, it is possible to use payload encryption on the application level to secure the transmitted information (without the need for full-fledged transport encryption).

8.5. Broker analysis

The two shortlisted brokers will now be analyzed in more detail and examined, to see whether it would be possible to implement the thesis with one or even both of them.

Ideally the broker supports the following key features:

- **TLS** As stated in Chapter 3, TLS is a required feature to eliminate the MITM.
- **Enhanced authentication** The enhanced authentication mechanism, using a challenge/response flow based on the new AUTH packets, must be supported to implement the authentication process mentioned in Chapter 3.
- **Publish / Subscribe interception** As mentioned in Chapter 4, every publish and subscribe activity of a client must be intercepted to check whether it is allowed to act on the given topic or not. Additionally, the client submits its claims over a predefined topic. Only through intercepting publish activity, the broker can listen for claims on this topic.
- **Message delivery interception** As the authorization will be implemented fully client-managed the authorization rules may change at runtime. Therefore, the broker needs to be able to check permission before every message delivery. It might be, that a claim owner updates a claim so that current subscriptions of clients on this topic become unauthorized.

8.5.1. HiveMQ

"Enterprise ready MQTT broker to move IoT data" is the claim which is prominent all over if one visits their website. For the longest time, only a paid version was available and the implementation was proprietary too. But since April 2019 the HiveMQ Community edition is available under the Apache 2.0 license free of charge. There is still a commercial version of HiveMQ available which mainly addresses features required by enterprises which are not part of the official MQTT specification.

The HiveMQ broker has an open source plugin system that allows it to hook into different events which occur within the broker. Additionally, the broker offers various callback interfaces that are straight forward to implement in custom plugins. The broker calls the plugin implementations at runtime.

Key feature analysis

The support of the defined key features was analyzed by consulting the documentation [26] and the source code [27] of HiveMQ.

Other security features

Permission based authentication HiveMQ provides a clean system to specify topic-based permissions. Restrictions of topics can be made for the following aspects:

- QoS
- Activity (Publish, Subscribe)
- Retain
- SharedSubscription
- SharedGroup
- Type (Allow, Deny)

As the approach in this thesis should be fully client-managed this feature could not be used directly. Nevertheless, it delivers valuable input how authorization features could be built.

Key feature	Supported	Explanation
TLS	Yes	TLS is fully supported (X.509 Certificates)
Enhanced authentication	No	This feature can not be found in the official HiveMQ documentation. In the HiveMQ community forum we found a thread [28] where a moderator stated, that HiveMQ is currently not supporting challenge/response authentication flows. Therefore this feature would have to be implemented by ourselves.
Publish / Subscribe interception	Yes	Both publish and subscribe interception can be achieved by implementing the interfaces *PublishAuthorizer* and *SubscriptionAuthorizer*. There is even a *PublishInboundInterceptor* interface which allows further modifications to application message parameters.
Message delivery interception	Yes	Message delivery can be intercepted by implementing the *PublishOutboundInterceptor* interface.

Table 8.3.: HiveMQ key feature analysis.

8.5.2. MQTTnet

MQTTnet is a MQTT-Broker implementation written in .NET Core and comes with a server and a client component. The Broker is MIT licensed which allows the use within open- and closed-source applications. The Broker accepts connections over TCP or Websocket. To make the connection secure, it can be established over TLS. For authentication, the username and password mechanism is available as specified by the MQTT protocol.

MQTTnet provides a clean builder-pattern based configuration approach, to inject several interface implementations to easily extend broker and client functionality.

Key feature analysis

The support of the defined key features was analyzed by consulting the documentation [29] and the source code [16] of MQTTnet.

Key feature	Supported	Explanation
TLS	Yes	TLS is fully supported (X.509 Certificates)
Enhanced authentication	No	Same as HiveMQ, MQTTnet is no fully MQTT 5.0 ready. In a GitHub issue [22] the repository owner stated, that the broker implementation is not ready to process authentication flows with AUTH packets.
Publish / Subscribe interception	Yes	Both publish and subscribe interception can be achieved by implementing the interfaces *IMqttServerApplicationMessageInterceptor* and *IMqttServerSubscriptionInterceptor*.
Message delivery interception	No	Intercepting outbound publish messages is not supported out of the box.

Table 8.4.: MQTTnet key feature analysis.

8.6. Decision

Based on our project hands-on, we were able to clone, build and debug both of our favorite brokers. For both implementations we found the respective entry points in order to implement an enhanced authentication mechanism which is specified in detail in the MQTT 5 specification.

At the end, we have decided to use MQTTnet according to the following arguments:

Known programming language and ecosystem As both project members are experienced C# developers, this fact argues in favor of taking the MQTTnet broker. Not only the programming language but also the ecosystem around C# is better known to the developers involved. This minimizes the risk of unforeseeable problems that are out of project scope.

Simplicity of the existing broker software As the project will implement a PoC, the underlying software which will be extended should be as simple as possible. HiveMQ is much more comprehensive in comparison to MQTTnet which is a disadvantage in this context. Additionally, both brokers do not support challenge/response authentication flows based on AUTH packets at evaluation time. After a short code analysis, the MQTTnet' code base seemed to be structured more simple. Therefore, the lightweight MQTTnet broker is preferred.

9. Requirements engineering

This chapter introduces the reader to the requirements we came up with. To present the requirements the form of use cases is used. They're perfectly suited to describe any functional requirements. To further clarify where the implemented authentication and authorization approach could be used, some user stories were defined as well.

9.1. Use case definitions

The use cases shown in Figure 9.1 provides an overview of the cases we're covering if a topic is unclaimed. In contrast, Figure 9.2 shows the cases which are implemented once a topic is claimed successfully.

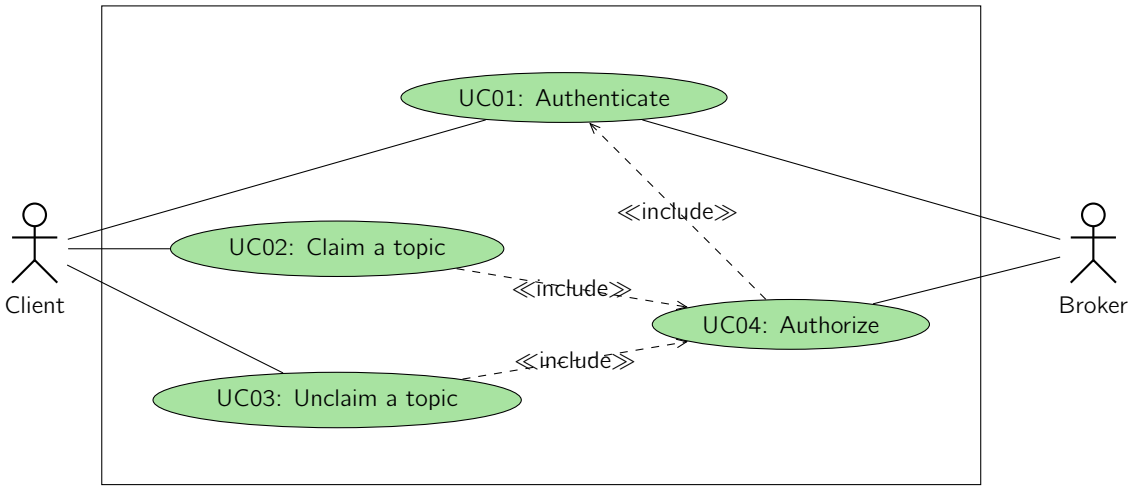


Figure 9.1.: Use cases to claim a topic.

9.1.1. UC01: Authenticate

Actors:	Client, Broker
Description:	The client and the broker perform an enhanced authentication according to Section 3.5
Trigger:	A client wants to claim, unclaim, publish or subscribe to a restricted topic
Preconditions:	-
Postconditions:	-
Normal Flow:	The client is authenticated successfully by the broker
Alternative Flow:	The client isn't authenticated and can just make use of the non restricted area of the broker
Exceptions:	-

Table 9.1.: UC01: Authenticate.

9.1.2. UC02: Claim a topic

Actors:	Client
Description:	Use case handles the process of claiming a topic
Trigger:	Broker receives a valid claim according to Table 6.2
Preconditions:	The Client has to be authenticated successfully, see Subsection 9.1.1
Postconditions:	The claim is created and stored by the broker
Normal Flow:	The broker parses and validates the received claim. If the data is valid the topic can be considered to be successfully claimed
Alternative Flow:	Parsing and validating failed. The data is dropped and the topic remains unclaimed
Exceptions:	A PUBACK response containing the reason code 0x00 (Success). In case the claim within the payload is invalid the reason code 0x99 (Payload format invalid) is returned.

Table 9.2.: UC01: Claim a topic.

9.1.3. UC03: Unclaim a topic

Actors:	Client
Description:	Use case handles the process of unclaiming a topic
Trigger:	Broker receives a valid unclaim according to Listing 6.9
Preconditions:	The Client has to be authenticated successfully, see Subsection 9.1.1
Postconditions:	The claim has been deleted by the broker
Normal Flow:	The broker parses and validates the unclaim request. If the data is valid, the topic is successfully unclaimed
Alternative Flow:	Parsing and validating failed. The data is dropped and the topic remains claimed
Exceptions:	A PUBACK response containing the reason code 0x00 (Success). If the topic to be unclaimed is not claimed on the broker, the reason code 0x99 (Payload format invalid) is returned.

Table 9.3.: UC03: Unclaim a topic.

9.1.4. UC04: Authorize

Actors:	Broker
Description:	The broker ensures the correct permission.
Trigger:	A client wants to claim/unclaim a topic
Preconditions:	The Client has to be authenticated successfully, see Subsection 9.1.1
Postconditions:	-
Normal Flow:	The broker ensures that any request made to the protected area is correctly authorized
Alternative Flow:	Permission is not granted, and the request canceled
Exceptions:	Depending on the case, the broker returns the following reason codes to clients within a PUBACK or a SUBACK packet. In case of authorized access the reason code 0x00 (Success, Granted QoS 0), 0x01 (Granted QoS 1) or 0x02 (Granted QoS 2) is returned. In case the broker finds a compromised claim, the reason code 0x83 (Implementation specific error) is returned. In case of unauthorized access the reason code 0x87 (Not authorized) is returned.

Table 9.4.: UC04: Authorize.

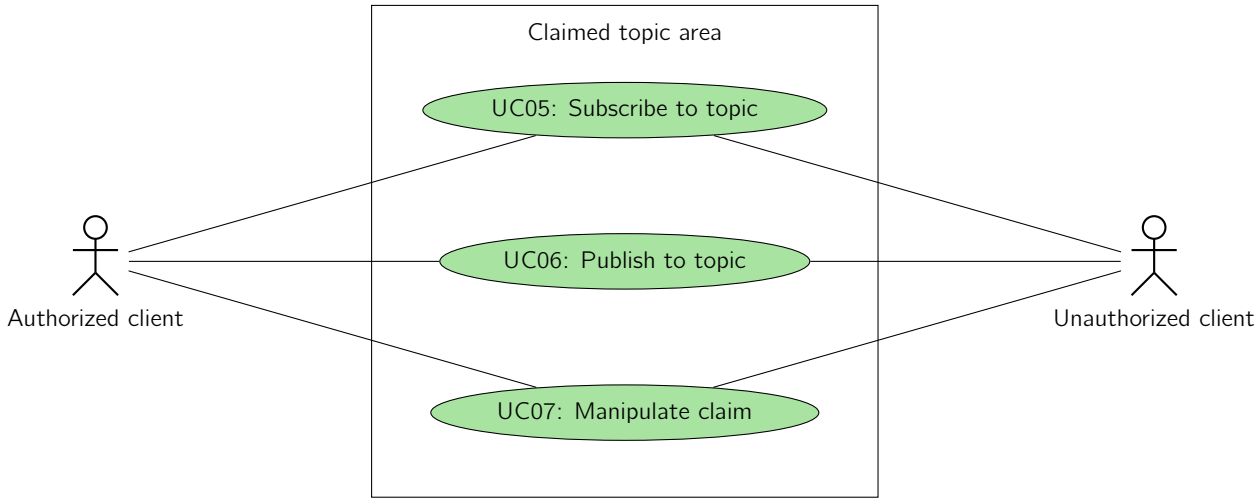


Figure 9.2.: Use cases after a topic is claimed.

9.1.5. UC05: Subscribe to a claimed topic

Actors:	Authorized client, Unauthorized client
Description:	Any client tries to subscribe to a claimed topic. Only if the client has the correct permissions, it will receive any messages from the topic.
Trigger:	A client subscribes to a topic
Preconditions:	The Client has to be authenticated successfully
Postconditions:	-
Normal Flow:	The broker ensures that only clients which are granted to the respective topic receive the messages
Alternative Flow:	No packets are received
Exceptions:	In case of authorized access a SUBACK response containing the reason code 0x0{0, 1, 2} (Granted QoS {0, 1, 2}). In case of an unauthorized access a SUBACK response containing the reason code 0x87 (Not authorized).

Table 9.5.: UC05: Subscribe to a claimed topic.

9.1.6. UC06: Publish to a claimed topic

Actors:	Authorized client, Unauthorized client
Description:	Any client tries to publish to a claimed topic. Only if the client has the correct permissions, it will be able to publish any messages to the topic.
Trigger:	A client publishes to a topic
Preconditions:	The Client has to be authenticated successfully
Postconditions:	-
Normal Flow:	The broker ensures that only clients which are granted to the respective topic can publish messages
Alternative Flow:	Received packets are dropped
Exceptions:	In case of authorized access, a PUBACK response containing the reason code 0x00 (Success). In case of an unauthorized access a PUBACK response containing the reason code 0x87 (Not authorized) is returned.

Table 9.6.: UC06: Publish to claimed topic.

9.1.7. UC07: Manipulate an existing claim

Actors:	Authorized client, Unauthorized client, Broker
Description:	Any client tries to update an existing claim. Only the owner of the claim can successfully modify the restriction.
Trigger:	A client publishes to a topic
Preconditions:	Access to the claim store was somehow obtained
Postconditions:	-
Normal Flow:	The broker ensures that only the owner of the client can manipulate an existing claim.
Alternative Flow:	-
Exceptions:	In case of authorized access, a PUBACK response containing the reason code 0x00 (Success). In case of an unauthorized access a PUBACK response containing the reason code 0x87 (Not authorized). In case of an invalid claim payload a PUBACK containing the reason code 0x99 (Payload format invalid).

Table 9.7.: UC07: Manipulate an existing claim.

9.2. User stories

The user stories described here are tailored to match the requirements the VAPE platform [30] came up with. Therefore, the SMOKER implementation perfectly suits as a backbone to the VAPE platform.

Note: The term "anyone" refers to any user which has a successfully authenticated MQTT session.

ID	Description
US01	As a VAPER, I want to create a private topic, where anyone can publish to, but is only readable by me.
US02	As a VAPER, I want to create a shared topic, where only invited users can publish to.
US03	As a VAPER, I want to create a shared topic, where only invited users can subscribe to.
US04	As a VAPER, I want to create a shared topic, where certain users are denied to subscribe.
US05	As a VAPER, I want to create a shared topic, where certain users are denied to publish.
US06	As a VAPER, I want to be able to delete an existing claim.
US07	As a VAPER, I want to be able to update an existing claim.

Table 9.8.: User stories.

In Chapter 10, acceptance tests will be derived from the user stories defined in Table 9.8.

9.3. Non-functional requirements

9.3.1. Documentation

Clean documentation of the code and the concepts we came up with are expected. This thesis shall be used to obtain this goal.

9.3.2. Performance

Performance is a critical part of any application built. In the world of IoT it's even more important because we have to deal with restricted processing power and having limited bandwidth is very likely. Securing a system always has its price. Nevertheless, the broker shall be as fast as possible and the payloads should be as small as possible. Performance tests will show the effective penalty we have to take into account.

9.3.3. Code quality

We refer to unit test to guarantee the functional correctness. Since this is just a PoC, no further code quality measures will be made.

9.3.4. Maintainability

The solution should be as easy to maintain as possible, therefore any changes made to existing open source projects will be committed back to the community. Code which is only used within this project has to be unit tested.

10. Testing

The goal of this chapter is to prove that the PoC meets all the requirements we have set in Chapter 9. Furthermore, this chapter not only focuses on functional correctness but also takes some non-functional requirements like performance and bandwidth goals into account.

Based on the requirements and user stories in Chapter 9, we can derive the following test cases.

- The client can successfully initiate an enhanced authentication flow.
- The broker can successfully handle an enhanced authentication.
- An authorized client can claim and unclaim a topic.
- A client can black- and white-list a certain topic.
- The persisted claim is immutable to any modifications done by any other than the owner.
- The client can subscribe to wildcard topics. Restrictions shall be checked, and if needed no delivery is made to the client if not granted to the specific topic.
- The challenge payload should be as small as possible.
- The clients need to compute the required keys as energy efficient as possible.

10.1. Unit testing

To test the functional requirements, unit tests are perfectly suited to guarantee the functional correctness of the individual parts. All tests are automatically executed as part of the Continuous Integration (CI) pipeline. All the tests are available in the *Mqttsec.Test* project which is part of the *SMOKER* solution. We didn't define any goals like having a test coverage of 80%, because this is a PoC and not meant to be used in production.

10.2. Manual testing

To make sure our implementation is conform to the protocol as specified by oasis [4], we can capture all the packets which are sent between the client and the broker. The tool Wireshark, a network analyzer, is used in order to achieve the capturing. If no authentication is enabled, one can see in Figure 10.1 the according connect packet, issued from the client to the broker. The broker then just answers with a CONNACK packet and the connection is established.

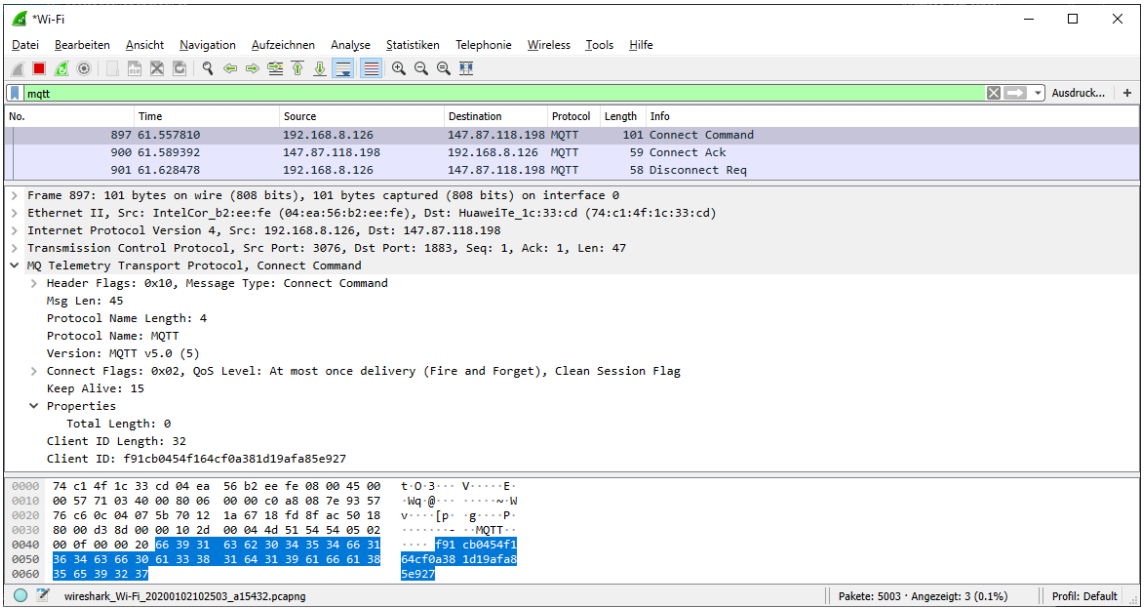


Figure 10.1.: Default MQTT client connection establishment.

In contrast, if we enable our authentication protocol we expect some AUTH packets before a CONNACK packet is sent. In Figure 10.2 we can see the "Continue authentication" (AUTH) packets exchange between the client and the broker. The packets sent matches the data specified in Section 3.5.

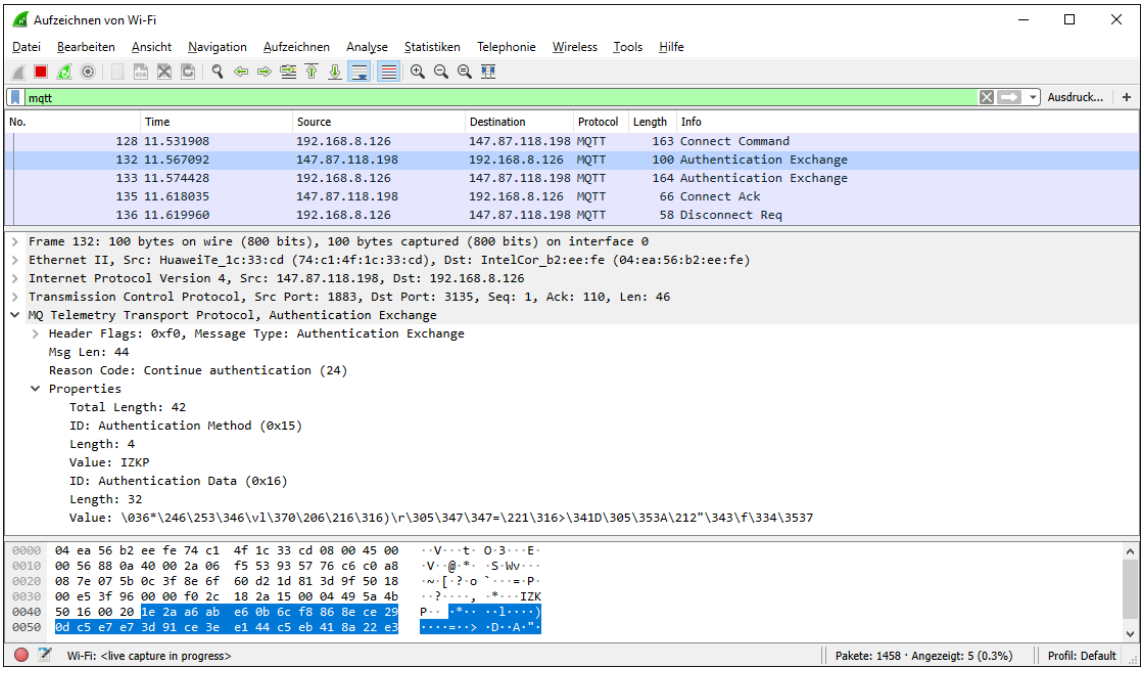


Figure 10.2.: Enhanced MQTT client connection establishment.

10.3. Performance testing

This section will benchmark the broker in order to get an idea of the penalty we have to take into account. Since we're dealing with a custom authentication protocol, we can't use an existing MQTT bench marking tool. Therefore, we have to write our own testing suite. The tools used to perform the benchmarks can be found as part of the solution in the *Smoker.Benchmark* project.

The MQTT protocol was built with performance in mind, according to MQTTnet [16] a single instance can process up to 70'000 messages / second (Tested on local machine (Intel i7 8700K) with MQTTnet client and server running in the same process using the TCP channel). Our tests are not running locally, we're more interested in a real world scenario where the broker is running on a dedicated Virtual Machine (VM). The client and the server are in the same network connected via a 5G Wi-Fi. To make the tests as deterministic as possible, the broker VM is limited to just one CPU core. Therefore, we expect a significantly lower number compared to MQTTnet, when it comes to messages / second.

For all tests, the following reference system is used:

Component	**Broker**	**Client**
CPU	Intel Core i7-3615QM CPU @ 2.30GHz x 1	Intel Core i7-8565U CPU @ 1.80GHz x 8
RAM	2GB	16GB
OS	Debian GNU/Linux 10 (buster)	Windows 10 pro

Table 10.1.: Reference system for testing.

10.3.1. Connections benchmark

The first benchmark is designed to give an indication of the maximum connections a client can establish per second.

#	**MQTTnet**	**Mosquitto**	**HiveMQ CE**	**MQTTnet**	**Mosquitto**	**HiveMQ CE**
		No Auth		SMOKER authentication		
01	293	273	229	185	182	x
02	298	286	259	185	185	x
03	345	280	271	191	189	x
04	287	275	242	210	190	x
05	270	264	238	207	191	x
06	308	265	238	216	189	x
07	294	263	241	200	193	x
08	314	260	99	194	193	x
09	347	256	73	206	191	x
10	313	258	217	200	192	x
Total	**3069**	**2680**	**2107**	**1994**	**1895**	x
Avg	**306.9**	**268**	**210.7**	**199.4**	**189.5**	x

Table 10.2.: Max. connections/sec the system can handle.

As one can see in Table 10.2, with enabled authentication (SMOKER) we loose approximately **35%** of performance in MQTTnet and about **29%** in mosquitto compared to no authentication at all. Since the authentication process is done only once per connection establishment, we consider this penalty as reasonable.

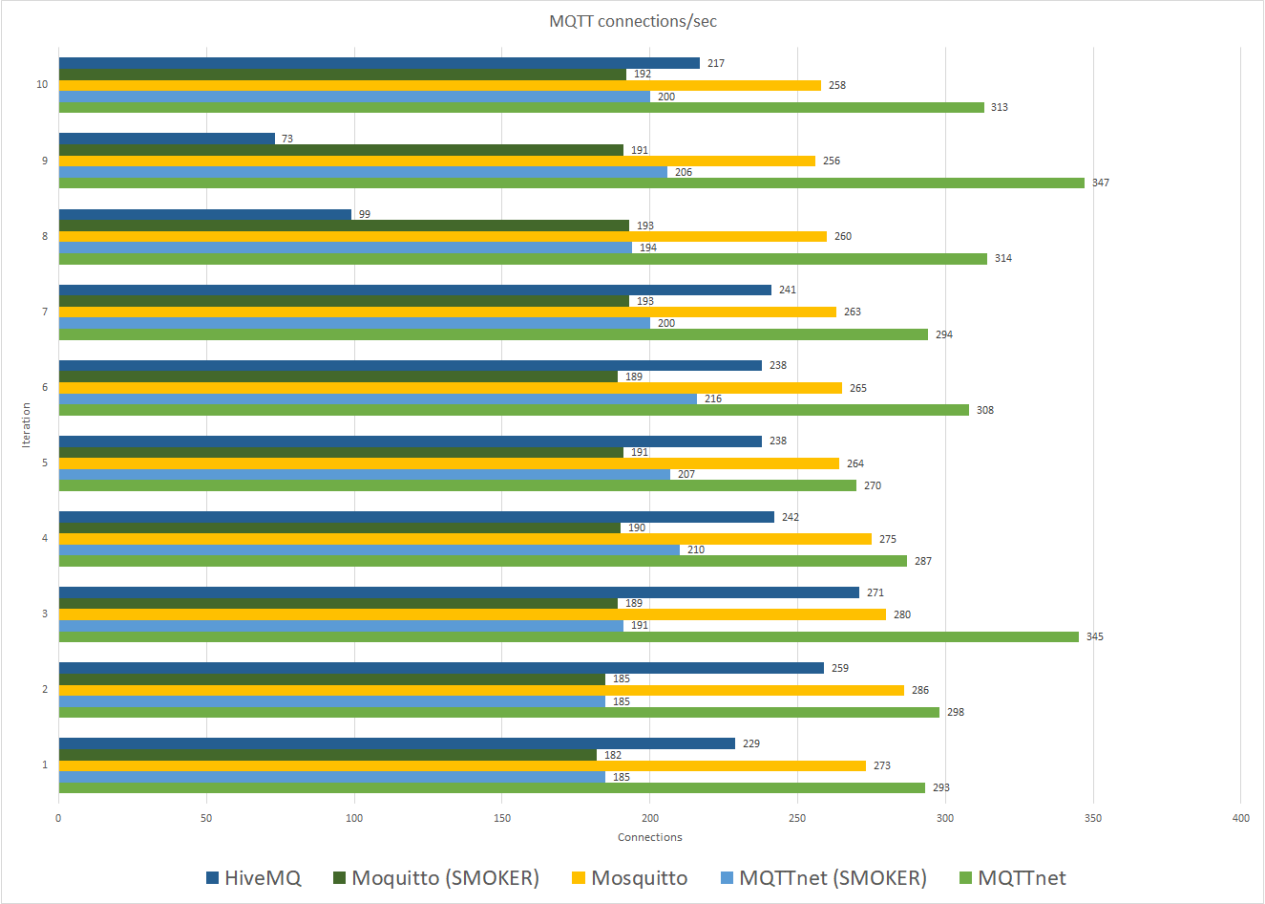


Figure 10.3.: Chart of Table 10.2 – higher is better.

10.3.2. Messages benchmark

The second benchmark is designed to give an indication of the maximum messages a client can publish per second.

As one can see in Table 10.3, with enabled authorization (SMOKER) we loose approximately **31%** of performance compared to no authorization at all. Taking into account that any publish call has to fetch the restriction from the database and evaluate the permissions it's quite understandable that those operations take it's time.

#	MQTTnet	Mosquitto	HiveMQ CE	MQTTnet	Mosquitto	HiveMQ CE
		No Auth		SMOKER authentication		
01	48557	52269	49596	33760	x	x
02	41210	52026	45578	39969	x	x
03	50772	50288	52453	31051	x	x
04	51604	46629	52106	30746	x	x
05	50118	44016	52273	33297	x	x
06	38974	50539	51705	35988	x	x
07	48355	52296	51363	20702	x	x
08	48331	50595	51791	35827	x	x
09	48469	48076	51374	28778	x	x
10	48206	49919	52223	36356	x	x
Total	**474596**	**496653**	**510462**	**326474**	x	x
Avg	**47459.6**	**49665.3**	**51046.2**	**32647.4**	x	x
Received	**464030**	**496653**	**510462**	**326474**	x	x
Lost	**10566**	**0**	**0**	**0**	x	x
Time	**13.00s**	**13.00s**	**49.12**	**52.13**	x	x

Table 10.3.: Max. messages/sec the system can handle.

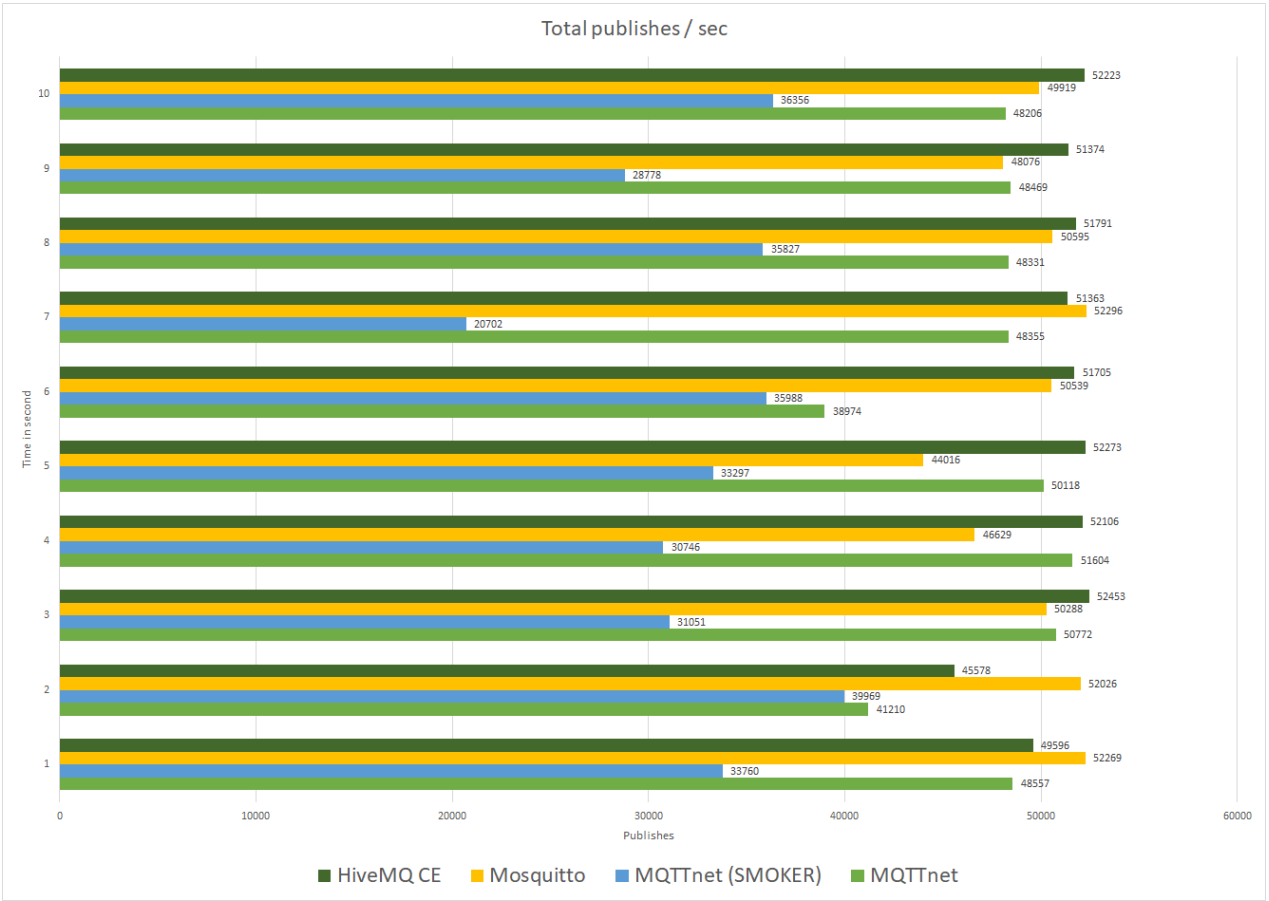


Figure 10.4.: Chart of Table 10.3 – higher is better.

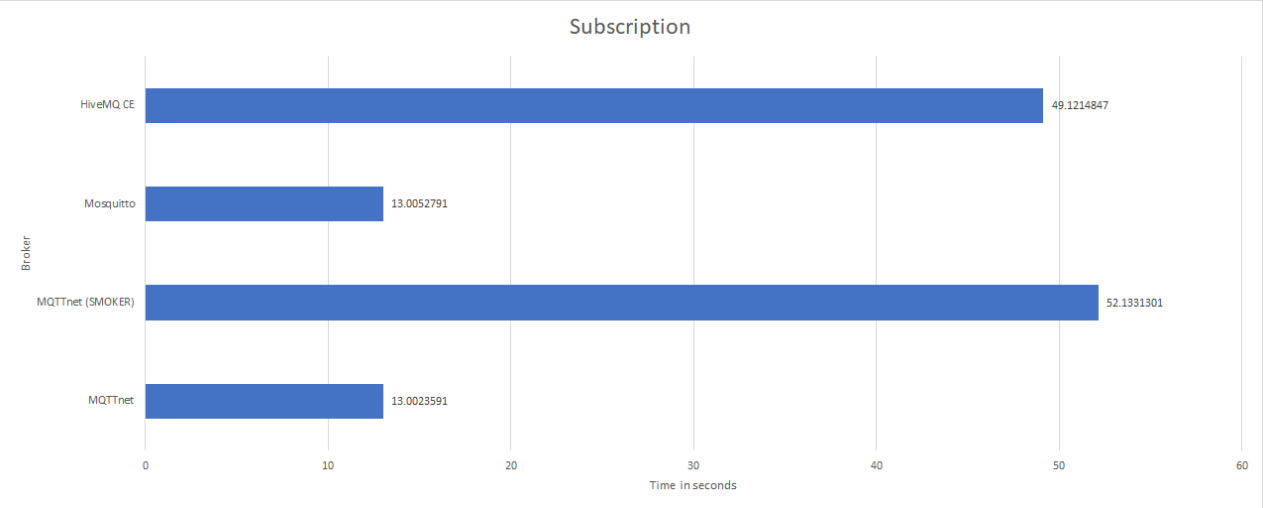


Figure 10.5.: Delivery time of Table 10.3 – lower is better.

As one can see in Figure 10.4 HiveMQ seems to be the fastest broker when it comes to packet acceptance. But they're all in a similar range. More important is the delivery time, the time a broker needs to dequeue the received packets and deliver it to the subscribed clients. Figure 10.5 shows de measurements of how long it took to receive all published packages. Mosquitto and MQTTnet are the clear winner regarding just the time, but taking into account that MQTTnet loses **10566** messages during this benchmark, it indicates that one shouldn't use MQTTnet for a productive environment. Indeed there are several issues on GitHub targeting this exact behavior. Although HiveMQ is not extended by the SMOKER authorization mechanism, the performance of packet delivery is very poor compared to the other contenders in this benchmark.

Brainstorming The current implementation of the SMOKER authorization extension is doing a database lookup for every publish to a restricted area (see Section 6.2 for further details). Usually permissions are relatively static and therefore should be cached. An in-memory cache could drastically reduce the slow lookups ase and increase the performance significantly. Eventually we could store the set of permissions for a given user during the authentication phase and store those is a client specific session.

10.3.3. Validating

ID	Description	Testresult
TC01	As a VAPER, I want to create a private topic, where anyone can publish to, but is only readable by me.	OK
TC02	As a VAPER, I want to create a shared topic, where only invited users can publish to.	OK
TC03	As a VAPER, I want to create a shared topic, where only invited users can subscribe to.	OK
TC04	As a VAPER, I want to create a shared topic, where certain users are denied to subscribe.	OK
TC05	As a VAPER, I want to create a shared topic, where certain users are denied to publish.	OK
TC06	As a VAPER, I want to be able to delete an existing claim.	OK
TC07	As a VAPER, I want to be able to update an existing claim.	OK
TC08	The client can successfully initiate an enhanced authentication flow	OK
TC09	The broker can successfully handle an enhanced authentication	OK
TC10	An authorized client can claim and un-claim a topic	OK
TC11	A client can black- and white-list a certain topic	OK
TC12	The persisted claim is immutable to any modifications done by any other than the owner	OK
TC13	The client can subscribe to wildcard topics. Restrictions shall be checked, and if needed no delivery is made to the client if not granted to the specific topic	NOK
TC14	The clients need to compute the required keys as energy efficient as possible	OK

Table 10.4.: All test cases with test results.

11. User Guide

This chapter describes how to set up a SMOKER and a corresponding client with the enhanced authentication and authorization approach introduced in this thesis. Note that this is running a PoC and is not a production ready solution.

11.1. Setup the broker

Depending on the brokers' use case, there are different components that need to be set up to get the desired environment running. The easiest way, is to run the provided docker image, which runs a SMOKER instance within an ASP.NET Core environment. The claim persistence can be configured to be in-memory or using a Mongo database. If a Mongo database is used, the database can be run within another docker container too. To get the docker containers running, make sure the target system has a running docker environment installed.

The SMOKER can also be run manually by checking out our code and compiling it by yourself. This approach also allows to run the broker without any ASP.NET Core environment around but thus without websocket support. To make the setup as easy as possible, only the docker way is documented in this chapter.

11.1.1. Configuration

The configuration is the same for the standalone and the ASP.NET Core version of the broker. The configuration is represented as a JSON-File which is stored under *{workdir}/Config/appsettings.json* and looks as follows:

```
{
    "MqttSec": {
        "Broker": {
            "Tcp": {
                "Ssl": false,
                "Port": 1883,
                "PortSsl": 8883
            },
            "Websocket": {
                "Enabled": true,
                "Port": 8000,
                "Path": "/mqtt"
            },
            "Database": {
                "ClaimsCollectionName": "Claims",
                "ConnectionString": "mongodb://root:example@mongo:27017",
                "DatabaseName": "mqttsec"
            }
        }
    }
    ... logging section excluded ...
}
```

Listing 11.1: Configuration example

Details to the above configuration properties from the *Mqttsec:Broker* section are described in the following Table 11.1:

Config property	Description
Tcp:Ssl	Enables the broker to use TLS
Tcp:Port	The TCP-Port to be used for the broker. The MQTT default port is 1883.
Tcp:PortSsl	The SSL TCP-Port to be used if TLS is enabled. The MQTT default SSL port is 8883.
Websocket	The websocket configuration section is ignored if the broker does not run within a ASP.NET Core environment.
Websocket:Enabled	Flag that enables Websocket if the broker runs within a ASP.NET Core environment.
Websocket:Port	The Websocket port where the HTTP server listens for Websocket connections.
Websocket:Path	The context path where the HTTP server listens for Websocket connections.
Database	The database configuration section is optional. If this section is not provided, the broker will implicitly use the In-Memory claim store which is described in Subsection 6.6.1.
Database:ClaimsCollectionName	The name of the entity where claims are hold if Mongo database is used as data store. If the collection does not exist, it will be created automatically.
Database:ConnectionString	The connection string to the Mongo database. The format is defined as follows *mongodb://{username}:{password}@{host}:{port}*.
Database:DatabaseName	The name of the Mongo database where the broker connects to. This database must be existent if the broker starts.

Table 11.1.: SMOKER configuration properties.

11.1.2. Setup the database

If the broker should be used with a Mongo database, such an instance must be set up. The simplest way to get a Mongo database instance running, may be starting a corresponding docker container. An example of such a container is delivered in Subsection 11.1.3. If you want to set it up natively on the target system, please refer to the Mongo DB web page for further instructions. Once the database is set up, make sure to configure the correct connection string within the configuration file.

11.1.3. Running using docker

To bring up a fully functional broker environment, a docker compose file provides remedy to fulfill this task. At least one container, the smoker, is needed. If the broker uses a Mongo database as data source, the corresponding container needs to be set up too. In this example the broker is used with a Mongo database.

The docker-compose file shown in Listing 11.2 brings up the needed containers. The broker itself and the Mongo database where the broker is connecting to.

```
version: '3'

services:
    smoker:
        restart: always
        depends_on:
            - mongo
        container_name: "smoker"
        image: registry.gitlab.ti.bfh.ch/mqttsec/mqttsec:latest
        volumes:
            - ~/work/docker/volumes/smoker/config:/app/Config
        ports:
            - "1883:1883"
            - "8080:8000"
        stdin_open: true
        tty: true

    mongo:
        image: mongo
        restart: always
        expose:
            - "27017"
        ports:
            - 27017:27017
        environment:
            MONGO_INITDB_ROOT_USERNAME: root
            MONGO_INITDB_ROOT_PASSWORD: example
```

Listing 11.2: Docker compose example

To make everything ready follow these steps:

1. Create a folder (e.g. *$HOME/work/docker/volumes/smoker/config*) to store the brokers' configuration, which will be mounted to the docker container.
2. Copy the content of Listing A.2 to a file named *appsettings.json* within the brokers' configuration folder and edit the configuration according to your needs.
3. Create a file named *docker-compose.yml* and paste the content of the Listing 11.2 in there and edit the configuration according to your needs. If no database is used the *mongo* service can be removed. Especially make sure the correct volume path for the broker configuration is set thus the configuration is loaded correctly.
4. On the console go to the folder where the *docker-compose.yml* file is stored and run *docker-compose up*.
5. Connect to the broker on the target system!

Checkout the Listing A.1 in the appendix for other helpful docker containers which were used to run the broker on a test system while developing.

11.2. Setup the client

Setting up a client is very straight forward as there is no need of additional configuration except setting up the client options and execute it within a .NET Core runtime environment.

```
1  var config = new ConfigManager();
2  var keyPair = config.ReadKeypair();
3
4  var extendedAuthClientHandler = new SmokerClientEnhancedAuthHandler(keyPair);
5  var encodedClientId = extendedAuthClientHandler.EncodeClientId();
6  config.StoreKeypair(keyPair);
7
8  // Create TCP based options using the builder.
9  var options = new MqttClientOptionsBuilder()
10     .WithAuthentication(Smoker.AuthenticationMethod,
           extendedAuthClientHandler.KeyPair.PublicKey)
11     .WithClientId(encodedClientId)
12     .WithCommunicationTimeout(TimeSpan.FromMinutes(2))
13     .WithExtendedAuthenticationExchangeHandler(extendedAuthClientHandler)
14     .WithProtocolVersion(MqttProtocolVersion.V500)
15     .WithTcpServer("localhost")
16     .Build();
17
18 // Create a new MQTT client.
19 var client = new MqttFactory().CreateMqttClient();
20 await client.ConnectAsync(options);
```

Listing 11.3: Client configuration

The *ConfigManager* helps loading and storing an already generated key pair as it might be reasonable to always use the same key for one client. Of course the key can individually be loaded from any other data source – however it must be a valid Ed25519 generated key as described in Chapter 5.

The lines 10 - 12 are enabling the enhanced SMOKER-Authentication. The clientID, which is set on line 11, must be retrieved from the *SmokerClientEnhancedAuthHandler::EncodeClientId* function as the broker needs this clientID to verify signatures provided by the client.

11.2.1. Claiming topics

Once the client is successfully connected to a running SMOKER instance, it can start claiming topics in its restricted area. To do this a claim must be built and published on the claim topic as shown in Listing 11.4:

```
1      // setup a restriction / claim
2      var rest = new Restriction(encodedClientId, "chat/private",
           RestrictionType.Blacklist);
3      rest.Permissions.Add(new Permission(AuthorizationConsts.AnyClientIdentifier,
           MqttAccessType.Subscribe));
4      var claim = new Claim(rest,
           enhancedAuthClientHandler.InitialKeyPairA.PrivateKey);
5
6      // claim
7      var claimResult = await client.ClaimAsync(claim);
```

Listing 11.4: Private chat topic claim

The *claimResult* variable on the last line can be consulted to verify whether the claim worked or not by checking the *ReasonCode* returned by the broker. Any other reason code than 0 (Success) has the meaning that something went wrong while claiming the topic.

The example in Listing 11.4 claims a topic which could be used for a private chat where everyone can publish, but only the claiming client can implicitly subscribe, as it is the owner. The *Restriction* constructor automatically enriches the topic to be claimed with needed information, like the restricted area prefix and the clientID as described in Subsection 6.4.2. The example would claim the following topic, where *{clientId}* substitutes the real clientID:

```
1    restricted/{clientId}/chat/private
```

Listing 11.5: Private chat topic example

11.2.2. Unclaiming topics

Once a topic is claimed, only the owner can unclaim it. To unclaim a topic, there is another extension method for the *IMqttClient* interface which makes it easy to unclaim:

```
1    // unclaim
2    var unclaimResult = await client.UnclaimAsync(claim.Restriction.TopicName);
```

Listing 11.6: Unclaim a topic

Same as claiming, the *UnclaimAsync* function returns the unclaim task which can be consulted to verify if the unclaim succeeded.

11.2.3. Usage hints

Keep claim state As MQTT is not a request / response protocol like HTTP, the client is not able to get all its current claims as long the broker does not provide read access to his data source. Therefore, it is recommended to keep state of its own claims on the client side. The current implementation does not offer such a feature.

12. Project management

12.1. Organization

Who	Role	Responsibilities
Dr. Reto Koenig	Tutor	Assessment based on scope and deliverables. Supervision of the thesis.
Dr. Federico Flueckiger	Expert	Assessment based on scope and deliverables.
Lukas Läderach & Cédric von Allmen	Students	Project management, Realization of the thesis, Communication with stakeholders

Table 12.1.: Involved project members.

12.1.1. Meetings

Date	Attendees	Location	Objectives
19.09.2019	Reto Koenig Lukas Läderach Cédric von Allmen	Online	Kick-Off, Discuss project structure
17.10.2019	Reto Koenig Lukas Läderach	Biel	Review milestones, Discuss tool chain, Discuss existing paper integration
04.11.2019	Federico Flueckiger Lukas Läderach Cédric von Allmen	Bern	Project introduction for the expert
08.11.2019	Reto Koenig Lukas Läderach Cédric von Allmen	Online	Discuss mathematical notations from algorithms
21.11.2019	Reto Koenig Lukas Läderach Cédric von Allmen	Biel	Discuss current implementation of Schnorr identification scheme, Solve mathematical problems of current implementation
19.12.2019	Reto Koenig Lukas Läderach Cédric von Allmen	Online	Discuss signature based authentication, Validate authorization schema, Discuss trust setting
06.01.2020	Reto Koenig Lukas Läderach Cédric von Allmen	Bern	Discuss book entry and document structure, Decide further actions regarding to the paper publication

Table 12.2.: Meeting protocol.

12.2. Development

This section describes how the project development was organized and which tools were selected to support the development process.

12.2.1. Process

As we are only two project members, we decided to organize the development process as easy as possible. Therefore, we did not choose a common approach (e.g. SCRUM or Waterfall) to manage the project. The project was organized with milestones and a backlog consisting of tasks where every task is assigned to a milestone. This entails an iterative approach where every project member just grabs an open task to implement.

12.2.2. Gitlab

The project source code is hosted on the BFH Gitlab instance. Gitlab offers several features that help to simplify the project management.

Code reviews with merge requests For every task which was grabbed from the backlog to implement a feature or write documentation, a feature-branch was created. After the work is done and the branch is ready to merge back to the master-branch, a Gitlab merge-request was created and assigned to the other project member. This merge request is then approved by the reviewer and merged back to the master-branch. This triggers the CI/CD pipeline for the master-branch.

Continuous integration To build and bundling the artifacts, the Gitlab CI environment was used. To following build stages were configured:

Stage	Description
Test	Runs the unit and integration tests within the .NET Core project for every pushed branch. If this stage fails, no further stages are triggered.
Build	Builds a new docker image only if the master-branch got pushed and publishes the new image to the project owned Docker registry. Additionally this document, written in LaTeX, is built for every branch. The built PDF document can be downloaded as an artifact over the Gitlab UI.

Table 12.3.: Gitlab build stages.

12.2.3. Deployment

To host a broker instance on an external system, a BFH virtual machine running Ubuntu 18.04.3 LTS is used. On this host the docker-compose file shown in Listing A.2 is running. This includes a Watchtower container, that checks the Gitlab docker registry for new images every 30 seconds. If a new broker image is detected, Watchtower automatically updates the image and restarts the broker container.

12.2.4. Milestones and actual working hours

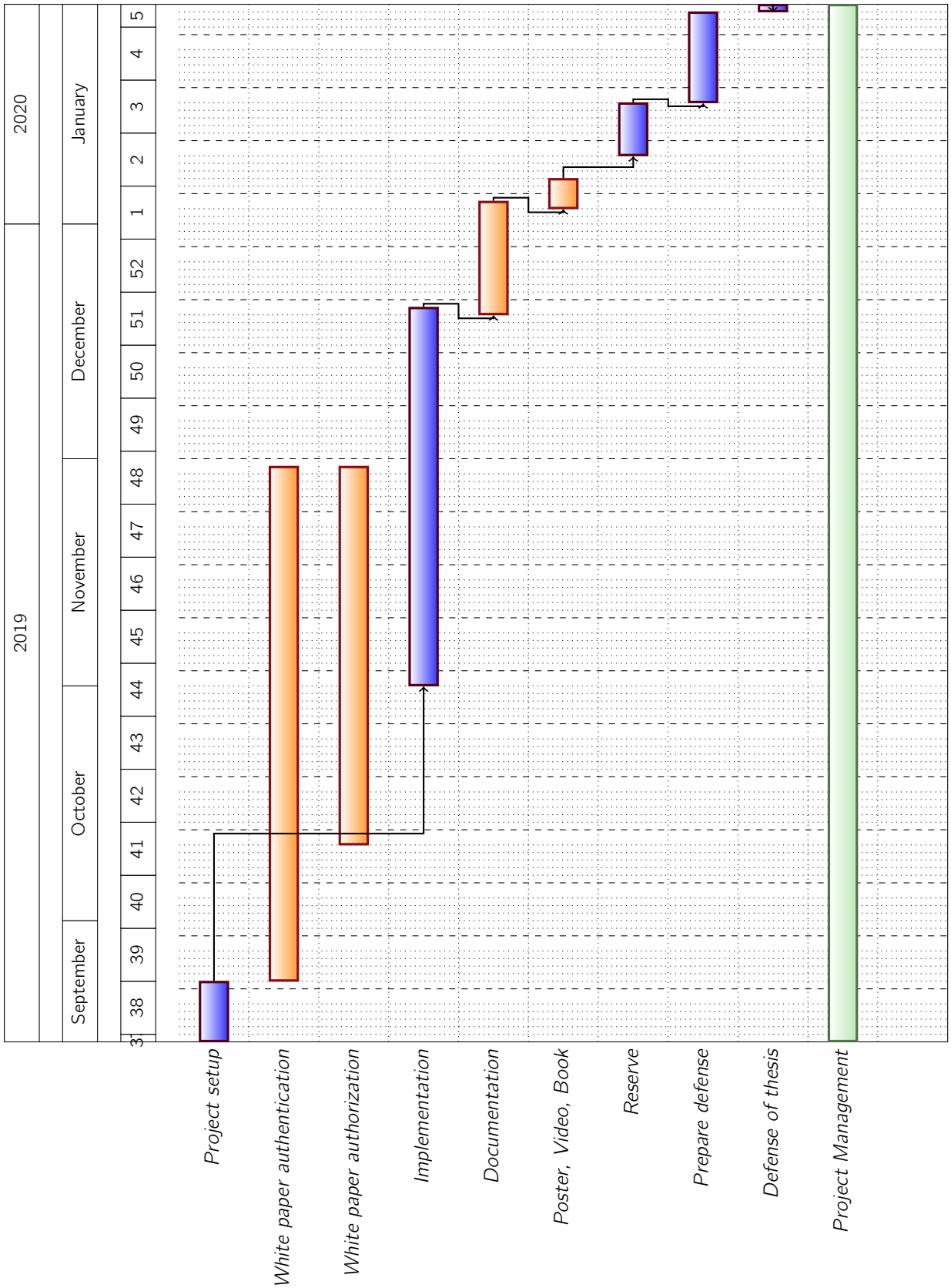
The following Table 12.4 is listing up the milestones with their duration and actual working hours. The working hours are summed up for both project members.

Milestone	From	To	Hours	Tasks
Project setup	15.09.2019	22.09.2019	24	Kick-Off, Setup project structure
White paper authentication	23.09.2019	29.11.2019	44	Finalize project 2 white paper
White paper authorization	11.10.2019	29.11.2019	46	Write white paper for client-managed authorization
Implementation	01.11.2019	20.12.2019	216	Implement authentication and authorization
Documentation	20.12.2019	03.01.2020	192	Document technical implementation, project management, conclusion
Poster, Video, Book	03.01.2020	06.01.2020	36	Create poster, video and book deliverables
Reserve	10.01.2020	16.01.2020	112	Finish documentation, bundle deliverables, final corrections
Prepare defense	17.01.2020	28.01.2020	48	Prepare the defense presentations, read documentation
Defense of thesis	29.01.2020	29.01.2020	8	Final defence of the thesis
Project Management	16.09.2019	16.01.2020	12	Manage project settings, meetings
Total	**136 days**		**738**	**5.42 hours per day**

Table 12.4.: Milestones and actual working hours.

A better visualization of the milestones over the project time line can be found in Subsection 12.2.5.

12.2.5. Gantt Chart



12.2.6. Tasks

Work items were tracked with Gitlab issues which can be found in the appendix Section B.1.

12.2.7. Working Together

As the agile method pretends, the backlog of tasks was processed one by one. If ever possible, difficult decisions or situations were discussed within the project team – if necessary, the problem was added to the agenda for the next meeting with the tutor, to find a solution.

Extreme Programming During the implementation phase of the thesis, the key features were implemented together at the same workstation, which brought more creativity and code quality. Further this coding approach helped to spread the technical know how to the project members.

12.3. Deliverables

Date	Deliverable
03.01.2020	Poster
07.01.2020	Book entry
16.01.2020	Thesis documentation, video, source code

Table 12.5.: Project deliverables.

12.4. Preliminary work

Within the scope of *Project 2* in the 8th semester, the theoretical aspects of the authentication mechanism was elaborated together with Dr. Reto Koenig which is the tutor of this thesis. Additionally, the first steps of the broker evaluation was started as MQTT 5.0 was just recently released at this time. It was therefore imperative to find a broker which implements the MQTT 5.0 specification as enhanced authentication was a required feature to realize this thesis.

Both the theoretical as well as the broker evaluation part, were not completely finished within *Project 2* as some more research work was needed. These two parts were finalized within this thesis in Chapter 3 (How to Authenticate MQTT Sessions Without Channel- and Broker Security) and Chapter 8 (Broker evaluation).

12.5. Project retrospective

Every work package done in this thesis could be assigned to one of the following categories:

- Development
- Documentation
- Project Management

Things succeed, others do not – in the end there are always some lessons learned. These learnings, whether in a positive or negative sense, are documented within this section.

12.5.1. Development

+ **Contributing to the open source world** As we have forked the MQTTNet broker and implemented missing MQTT 5.0 features, we created a pull-request [23] to the upstream repository. Fortunately repository contributors and even the repository owner reviewed our code and made suggestions for improvements. The force of the open source mindset was therefore for our benefit. Hopefully we could deliver input for the enhanced authentication implementation which is currently missing in MQTTNet.
+ **Extreme Programming** In particular we followed the aspect of pair-programming which ensured that the technical know how of the implementation was spread over the project members. Furthermore, pair-programming encouraged creativity as well as the fun factor.
– **Don't implement your own crypto** In the first phase of the implementing milestone, we decided to implement the Schnorr NIZKP algorithm by ourselves. As we are actually not cryptography students, this became pretty messy. We had difficulties with understanding and interpreting the basic cryptography notations used in papers and RFC's. Hence, we decided to follow the common cryptography Mantra "*Don't implement your own crypto*" and started using well proven libraries.

12.5.2. Documentation

+ **The use of LaTeX** As both project members are experienced programmers, the use of LaTeXwas predestined. The plain text format made it easy to integrate the document into version control and continuous integration environments. Valuable time could also be saved on formatting the document as a lot of components are looking good out of the box.

12.5.3. Project management

– **Time reporting** The initial euphoria about the topic has led to the fact that time reporting was not reliably conducted from the beginning or has even been forgotten. The retroactive derivation of time expenditure was therefore very laborious and time-consuming.

13. Future work

This chapter gives you an overview of possible tasks which could/should be done in order to achieve maximum community acceptance and push the concepts shown in this thesis towards a standard.

13.1. Code

As mentioned several times, the code created during this thesis isn't production ready. The most critical parts are unit tested, though. The broker we choose allowed for speedy development cycles because C# is the programming language we are most familiar with.

To achieve the best performance, a more commonly used broker should probably be used. Mosquitto seems to be one of the fastest and efficient broker available. Due to the fact, that it's written in C, a low memory footprint, efficient resource usage and high portability is achieved. It also features a plugin architecture, which should allow to implement the concepts shown in this thesis relatively easily. More info can be found Subsection 5.6.3.

13.2. Paper

This thesis basically introduced two independent concepts:

- How to Authenticate MQTT Sessions Without Channel- and Broker Security
- Client-Managed topic-based authorization

The first concept is already extracted to a paper, which is published on arxiv.org [10]. The second concept is only part of this thesis. The next logical step would be to extract a paper from it and publish it on arxiv.org as well. As always in security related topics, the more people reviewing and commenting on the concepts shown, the smaller the possibility of potential flaws.

13.3. Community

Unfortunately, even if one is able to create a technological bullet proof solution, it doesn't guarantee a wide adaption. Politics and community acceptance is as important as having a rock solid technical solution. Therefore, one should pick important IoT related conferences and start promoting the technology. Once enough people might be convinced that the solutions shown are a great enhancement, there might be a small chance to make it into the MQTT specification.

14. Conclusion

With the version 5.0, MQTT has made a huge step forward in order to secure its position as one of the leading protocols used within IoT. Especially for developers, the new standard provides some valuable additions which simplifies several tasks. On one hand it's a bit unfortunate that the protocol still doesn't provide a standard for authentication and authorization. On the other hand, the protocol is designed to give max. flexibility to achieve almost anything one can think of.

According to Gartner [31], in 2018 there have been over 10 billion internet-connected things out there. This number is expected to increase about 32% in 2019. We expect this trend to continue in the upcoming years. Therefore, having strong, secure and fast security in place is mandatory.

Our approach not only takes the security aspect to one of the latest achievements made in science, but also promotes secure resource sharing by introducing a full client managed authorization scheme. This is an important aspect considering the costs involved by maintaining a reliable infrastructure.

Retrospectively, the broker chosen during evaluation might not be ideal for productive environments, it was a solid base to implement the PoC though. However, from our perspective the implementation for Mosquitto should be finished, since it's still the most efficient broker when it comes to CPU and memory usage. The benchmarking suite, which was also developed during this thesis, lays down a good foundation in order to further improve the concepts or just to make other brokers SMOKER ready.

The resulting SMOKER implementation can be considered as a success. All the goals defined by the thesis advisor were reached. Especially the client-managed authorization part was not only introduced but also implemented.

Declaration of primary authorship

I / We hereby confirm that I / we have written this thesis independently and without using other sources and resources than those specified in the bibliography. All text passages which were not written by me are marked as quotations and provided with the exact indication of its origin.

Place, Date:	Bern, March 28, 2020	
Last Name/s, First Name/s:	Lukas Läderach	Cédric von Allmen
Signature/s:	.....	.....

Glossary

.NET Core .NET Core is a free and open-source, managed computer software framework for Windows, Linux, and macOS operating systems. It is a cross-platform successor to .NET Framework. The project is primarily developed by Microsoft and released under the MIT License..

Access Control List (ACL) An ACL with respect to a computer file system, is a list of permissions attached to an object. An ACL specifies which users or system processes are granted access to objects, as well as what operations are allowed on given objects. Each entry in a typical ACL specifies a subject and an operation. For instance, if a file object has an ACL that contains (Alice: read,write; Bob: read), this would give Alice permission to read and write the file and Bob to only read it. .

Continuous Integration (CI) Continuous integration (CI) is the practice of automating the integration of code changes from multiple contributors into a single software project. The CI process is comprised of automatic tools that assert the new code's correctness before integration. A source code version control system is the crux of the CI process. The version control system is also supplemented with other checks like automated code quality tests, syntax style review tools, and more. .

Elliptic Curve Cryptography (ECC) Elliptic-curve cryptography (ECC) is an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields. ECC requires smaller keys compared to non-EC cryptography to provide equivalent security. .

Elliptic Curve Discrete Logarithm Problem (ECDLP) Public-key cryptography is based on the intractability of certain mathematical problems. Early public-key systems are secure assuming that it is difficult to factor a large integer composed of two or more large prime factors. For elliptic-curve-based protocols, it is assumed that finding the discrete logarithm of a random elliptic curve element with respect to a publicly known base point is infeasible: this is the "elliptic curve discrete logarithm problem" (ECDLP). The security of elliptic curve cryptography depends on the ability to compute a point multiplication and the inability to compute the multiplicand given the original and product points. The size of the elliptic curve determines the difficulty of the problem. The primary benefit promised by elliptic curve cryptography is a smaller key size, reducing storage and transmission requirements, i.e. that an elliptic curve group could provide the same level of security afforded by an RSA-based system with a large modulus and correspondingly larger key: for example, a 256-bit elliptic curve public key should provide comparable security to a 3072-bit RSA public key. .

Edwards-curve Digital Signature Algorithm (EdDSA) In public-key cryptography, Edwards-curve Digital Signature Algorithm (EdDSA) is a digital signature scheme using a variant of Schnorr signature based on Twisted Edwards curves. It is designed to be faster than existing digital signature schemes without sacrificing security. It was developed by a team including Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. The reference implementation is public domain software. .

Hypertext Transfer Protocol (HTTP) The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web, where hypertext documents include hyperlinks to other resources that the user can easily access, for example by a mouse click or by tapping the screen in a web browser. .

Internet of Things (IoT) Is the network of devices, vehicles, and home appliances that contain electronics, software, actuators, and connectivity which allows these things to connect, interact and exchange data. .

Machine to Machine (M2M) Machine to machine (M2M) is direct communication between devices using any communications channel. Machine to machine communication can include industrial instrumentation, enabling a sensor or meter to communicate the information it records (such as temperature, inventory level, etc.) to application software that can use it (for example, adjusting an industrial process based on temperature or placing orders to replenish inventory). Such communication was originally accomplished by having

a remote network of machines relay information back to a central hub for analysis, which would then be rerouted into a system like a personal computer. .

Man In The Middle (MITM) In cryptography and computer security, a man-in-the-middle attack (MITM) is an attack where the attacker secretly relays and possibly alters the communications between two parties who believe that they are directly communicating with each other. .

Message Queuing Telemetry Transport (MQTT) Is an ISO standard publish-subscribe-based messaging protocol. It works on top of the TCP protocol and is designed for connections with remote locations where a "small code footprint" is required or the network bandwidth is limited. The publish-subscribe messaging pattern requires a message broker. .

National Institute of Standards and Technology (NIST) The National Institute of Standards and Technology (NIST) is a physical sciences laboratory, and a non-regulatory agency of the United States Department of Commerce. Its mission is to promote innovation and industrial competitiveness. .

Node Package Manager (NPM) npm (Node Package Manager) is a package manager for the JavaScript programming language. It is the default package manager for the JavaScript runtime environment Node.js. .

Proof of Concept (PoC) Proof of concept (PoC) is a realization of a certain method or idea in order to demonstrate its feasibility, or a demonstration in principle with the aim of verifying that some concept or theory has practical potential. A proof of concept is usually small and may or may not be complete. .

Physical Unclonable Function (PUF) A physical unclonable function (sometimes also called physically unclonable function), or PUF, is a physical object that for a given input and conditions (challenge), provides a physically-defined "digital fingerprint" output (response) that serves as a unique identifier, most often for a semiconductor device such as a microprocessor. PUFs are most often based on unique physical variations which occur naturally during semiconductor manufacturing. .

Quality of Service (QoS) The Quality of Service (QoS) level is an agreement between the sender of a message and the receiver of a message that defines the guarantee of delivery for a specific message. There are 3 QoS levels in MQTT: At most once (0), At least once (1), Exactly once (2). .

Schnorr Non-Interactive Zero-Knowledge Proof (Schnorr NIZKP) The Schnorr NIZK proof allows one to prove the knowledge of a discrete logarithm without leaking any information about its value. It can serve as a useful building block for many cryptographic protocols to ensure that participants follow the protocol specification honestly. .

Transmission Control Protocol (TCP) The Transmission Control Protocol (TCP) is one of the main protocols of the Internet protocol suite. It originated in the initial network implementation in which it complemented the Internet Protocol (IP). Therefore, the entire suite is commonly referred to as TCP/IP. TCP provides reliable, ordered, and error-checked delivery of a stream of octets (bytes) between applications running on hosts communicating via an IP network. Major internet applications such as the World Wide Web, email, remote administration, and file transfer rely on TCP. .

Transport Layer Security (TLS) Transport Layer Security, are cryptographic protocols designed to provide communications security over a computer network. Several versions of the protocols find widespread use in applications such as web browsing, email, instant messaging. Websites can use TLS to secure all communications between their servers and web browsers. .

Virtual Machine (VM) In computing, a virtual machine (VM) is an emulation of a computer system. Virtual machines are based on computer architectures and provide functionality of a physical computer. .

Virtual Private Network (VPN) A virtual private network extends a private network across a public network, and enables users to send and receive data across shared or public networks as if their computing devices were directly connected to the private network. Applications running on a computing device, e.g., a laptop, desktop, smartphone, across a VPN may therefore benefit from the functionality, security, and management of the private network. .

ASP.NET Core ASP.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud-based, Internet-connected applications..

Base32 Base32 is one of several base 32 transfer encodings. Base32 uses a 32-character set comprising the twenty-six upper-case letters A–Z, and the digits 2–7. Each Base32 digit represents exactly 5 bits of data. Three 8-bit bytes (i.e., a total of 24 bits) ca therefore be represented by five 5-bit Base32 digits..

Base64 In computer science, Base64 is a group of binary-to-text encoding schemes that represent binary data in an ASCII string format by translating it into a radix-64 representation. Each Base64 digit represents exactly 6 bits of data. Three 8-bit bytes (i.e., a total of 24 bits) can therefore be represented by four 6-bit Base64 digits..

builder The builder pattern is a design pattern designed to provide a flexible solution to various object creation problems in object-oriented programming. The intent of the Builder design pattern is to separate the construction of a complex object from its representation..

Curve25519 In cryptography, Curve25519 is an elliptic curve offering 128 bits of security and designed for use with the elliptic curve Diffie–Hellman key agreement scheme. It is one of the fastest ECC curves and is not covered by any known patents. The reference implementation is public domain software..

Docker Docker is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels..

Ed25519 Ed25519 is the EdDSA signature scheme using SHA-512 (SHA-2) and Curve25519. Furthermore the team around Bernstein has optimized Ed25519 for the x86-64 Nehalem/Westmere processor family. Verification can be performed in batches of 64 signatures for even greater throughput. Ed25519 is intended to provide attack resistance comparable to quality 128-bit symmetric ciphers. Public keys are 256 bits in length and signatures are twice that size..

ephemeral Ephemerality is the concept of things being transitory, existing only briefly. Typically the term ephemeral is used to describe objects found in nature, although it can describe a wide range of things, including human artifacts intentionally made to last for only a temporary period, in order to increase their perceived aesthetic value..

Extreme Programming Extreme Programming (XP) is an agile software development framework that aims to produce higher quality software, and higher quality of life for the development team. XP is the most specific of the agile frameworks regarding appropriate engineering practices for software development. Other elements of extreme programming include: programming in pairs or doing extensive code review, unit testing of all code, avoiding programming of features until they are actually needed, a flat management structure, code simplicity and clarity, expecting changes in the customer's requirements as time passes and the problem is better understood, and frequent communication with the customer and among programmers..

Gitlab CI GitLab CI/CD is a tool built into GitLab for software development through the continuous methodologies: Continuous Integration (CI), Continuous Delivery (CD) and Continuous Deployment (CD).

libsodium Sodium is a modern, easy-to-use software library for encryption, decryption, signatures, password hashing and more. It is a portable, cross-compilable, installable, packable fork of NaCl, with a compatible API, and an extended API to improve usability even further..

Mongo database MongoDB is a cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with schema. MongoDB is developed by MongoDB Inc. and licensed under the Server Side Public License (SSPL)..

MQTT.js MQTT.js is a client library for the MQTT protocol, written in JavaScript for node.js and the browser..

NaCl NaCl (pronounced "salt") is a new easy-to-use high-speed software library for network communication, encryption, decryption, signatures, etc. NaCl's goal is to provide all of the core operations needed to build higher-level cryptographic tools. Of course, other libraries already exist for these core operations. NaCl advances the state of the art by improving security, by improving usability, and by improving speed. The core NaCl development team consists of Daniel J. Bernstein (University of Illinois at Chicago and Technische Universiteit Eindhoven), Tanja Lange (Technische Universiteit Eindhoven), and Peter Schwabe (Radboud Universiteit Nijmegen)..

nonce In cryptography, a nonce is an arbitrary number that can be used just once in a cryptographic communication. It is similar in spirit to a nonce word, hence the name. It is often a random or pseudo-random number issued in an authentication protocol to ensure that old communications cannot be reused in replay attacks. They can also be useful as initialization vectors and in cryptographic hash functions..

publish/subscribe In software architecture, publish–subscribe is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead categorize published messages into classes without knowledge of which subscribers, if any, there may be. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are..

replay attack An attack on a security protocol using replay of messages from a different context into the intended (or original and expected) context, thereby fooling the honest participant(s) into thinking they have successfully completed the protocol run..

session hijacking In computer science, session hijacking is the exploitation of a valid computer session, sometimes also called a session key, to gain unauthorized access to information or services in a computer system..

SHA-256 SHA-256 is a member of the SHA-2 cryptographic hash functions designed by the NSA. SHA stands for Secure Hash Algorithm. Cryptographic hash functions are mathematical operations run on digital data; by comparing the computed "hash" (the output from execution of the algorithm) to a known and expected hash value, a person can determine the data's integrity. A one-way hash can be generated from any piece of data, but the data cannot be generated from the hash..

Sodium.Core libsodium-core, or better said, libsodium for .NET Core is a .net standard 2.0 compliant fork libsodium-net. it is basically a C# wrapper around libsodium. For those that don't know, libsodium is a portable implementation of Daniel Bernstein's fantastic NaCl library..

Watchtower A process for watching your Docker containers and automatically restarting them whenever their base image is refreshed..

Websocket WebSocket is a computer communications protocol, providing full-duplex communication channels over a single TCP connection. The WebSocket protocol was standardized by the IETF as RFC 6455 in 2011, and the WebSocket API in Web IDL is being standardized by the W3C..

Bibliography

[1] Hivemq. [Online]. Available: https://www.hivemq.com/

[2] BehrTech. (2019) Basic mqtt network topology. (Accessed on 01/02/2020). [Online]. Available: https://behrtech.com/wp-content/uploads/2019/03/MQTT-.jpg

[3] HiveMQ-Team. (2019, August) Mqtt topics and best practices - mqtt essentials: Part 5. (Accessed on 12/16/2019). [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices/

[4] A. Banks, E. Briggs, K. Borgendale, and R. Gupta. (2019, March) Mqtt version 5.0. (Accessed on 09/27/2019). [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

[5] Json - wikipedia. (Accessed on 11/19/2019). [Online]. Available: https://en.wikipedia.org/wiki/JSON

[6] C. Paar, J. Pelzl, and B. Preneel, *Understanding Cryptography - A Textbook for Students and Practitioners*, 1st ed. Berlin Heidelberg: Springer Science and Business Media, 2009.

[7] Public-key cryptography - wikipedia. (Accessed on 10/16/2019). [Online]. Available: https://en.wikipedia.org/wiki/Public-key_cryptography

[8] Authentication - wikipedia. (Accessed on 09/27/2019). [Online]. Available: https://en.wikipedia.org/wiki/Authentication

[9] Authorization - wikipedia. (Accessed on 09/27/2019). [Online]. Available: https://en.wikipedia.org/wiki/Authorization

[10] R. E. Koenig, L. Läderach, and C. von Allmen, "[1904.00389] how to authenticate mqtt sessions without channel- and broker security," (Accessed on 12/26/2019). [Online]. Available: https://arxiv.org/abs/1904.00389

[11] F. Hao, "Schnorr non-interactive zero-knowledge proof," RFC Editor, RFC 8235, September 2017, (Accessed on 09/27/2019). [Online]. Available: https://tools.ietf.org/rfc/rfc8235.txt

[12] P. Matias, P. Barbosa, T. Cardoso, D. Mariano, and D. Aranha, "[1708.05844] nizkctf: A non-interactive zero-knowledge capture the flag platform," pp. 6–8, August 2017, (Accessed on 12/16/2019). [Online]. Available: https://arxiv.org/abs/1708.05844

[13] D. Giry. (2008) Keylength – cryptographic key length recommendation. (Accessed on 09/27/2019). [Online]. Available: http://www.keylength.com

[14] Eclipse mosquitto. [Online]. Available: https://mosquitto.org/

[15] HiveMQ-Team. (2015, May) Authorization - mqtt security fundamentals. (Accessed on 12/16/2019). [Online]. Available: https://www.hivemq.com/blog/mqtt-security-fundamentals-authorization/

[16] MQTTnet. (2019, December) Mqttnet: Mqttnet is a high performance .net library for mqtt based communication. (Accessed on 12/23/2019). [Online]. Available: https://github.com/chkr1011/MQTTnet

[17] S. Lanthemann, "Mqtt-fce," Thesis, Januar 2016.

[18] D. J. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters, "Twisted edwards curves," Cryptology ePrint Archive, Report 2008/013, 2008. [Online]. Available: https://eprint.iacr.org/2008/013

[19] ——. (2017) Ed25519 - high-speed high-security signatures. [Online]. Available: https://ed25519.cr.yp.to

[20] S. Krempl. (2010, December) 27c3: Sicherheitssystem der playstation 3 ausgehebelt. (Accessed on 01/16/2020). [Online]. Available: https://www.heise.de/security/meldung/27C3-Sicherheitssystem-der-Playstation-3-ausgehebelt-1161876.html

[21] Github. [Online]. Available: https://www.github.com/

[22] jimsch. (2019, May) Issue #641 - version 5.0 authentication examples. (Accessed on 12/23/2019). [Online]. Available: https://github.com/chkr1011/MQTTnet/issues/641

[23] L. Läderach and C. von Allmen. (2019, December) Enhanced authentication for mqttnet. (Accessed on 12/23/2019). [Online]. Available: https://github.com/chkr1011/MQTTnet/pull/835

[24] ——. (2019, November) Enhanced authentication for mqtt.js. [Online]. Available: https://github.com/quickstar/MQTT.js/tree/topic/implement-enhanced-auth

[25] B. Krebs. (2017, January) Beating json performance with protobuf. (Accessed on 01/12/2020). [Online]. Available: https://auth0.com/blog/beating-json-performance-with-protobuf/

[26] Hivemq user guide. [Online]. Available: https://www.hivemq.com/docs/4.2/hivemq/introduction.html

[27] Hivemq source code. [Online]. Available: https://github.com/hivemq

[28] JakubS. (2019, September) Auth packet support for custom authentication. (Accessed on 10/08/2020). [Online]. Available: https://community.hivemq.com/t/auth-packet-support-for-custom-authentication/77

[29] MQTTnet. Mqttnet wiki documentation. (Accessed on 09/27/2019). [Online]. Available: https://github.com/chkr1011/MQTTnet/wiki

[30] J. Baiutti and S. P. Wittwer, "Vape platform," (Accessed on 12/30/2019).

[31] C. Pettey. (2018) Unlock iot success by identifying and empowering the iot architect. [Online]. Available: https://www.gartner.com/smarterwithgartner/the-emergence-of-the-iot-architect/

List of Figures

List of Tables

List of listings

APPENDICES

A. Implementation

A.1. Docker deployment

```
version: '3'

services:

  # Watchtower
  watchtower:
    restart: always
    image: containrrr/watchtower
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    environment:
      - REPO_USER=svc_microfast
      - REPO_PASS=microfast1234!
    command: --interval 30 --debug --label-enable

  # Start MQTTSec Broker
  mqttsec.broker:
    restart: always
    depends_on:
      - mongo
    container_name: "mqttsec.broker"
    image: registry.gitlab.ti.bfh.ch/mqttsec/mqttsec:latest
    ports:
      - "1883:1883"
      - "8080:8000"
    stdin_open: true
    tty: true
    labels:
      - com.centurylinklabs.watchtower.enable=true

  # Reverseproxy fÃijr Portainer
  reverseproxy:
    restart: always
    image: jwilder/nginx-proxy
    ports:
      - 80:80
      - 443:443
    volumes:
```

```
      - /nginx/certs:/etc/nginx/certs:ro
      - reverseproxy_data:/etc/nginx/vhost.d
      - reverseproxy_data:/usr/share/nginx/html
      - /var/run/docker.sock:/tmp/docker.sock:ro
    labels:
      com.github.jrcs.letsencrypt_nginx_proxy_companion.nginx_proxy: ""

  # Certbot fÃijr SSL-Zertifikate
  certbot:
    restart: always
    depends_on:
      - reverseproxy
    image: jrcs/letsencrypt-nginx-proxy-companion
    volumes:
      - /nginx/certs:/etc/nginx/certs:rw
      - reverseproxy_data:/etc/nginx/vhost.d
      - reverseproxy_data:/usr/share/nginx/html
      - /var/run/docker.sock:/var/run/docker.sock:ro

  # Portainer
  portainer:
    restart: always
    depends_on:
      - certbot
      - reverseproxy
    image: portainer/portainer
    expose:
      - "9000"
    command: -H unix:///var/run/docker.sock
    volumes:
      - portainer_data:/data portainer/portainer
      - /var/run/docker.sock:/var/run/docker.sock
    environment:
      VIRTUAL_HOST: portainer.mqttsec.microfast.ch
      LETSENCRYPT_HOST: portainer.mqttsec.microfast.ch
      LETSENCRYPT_EMAIL: cnvonallmen@hotmail.com

  # Mongo DB
  mongo:
    image: mongo
    restart: always
    expose:
      - "27017"
    ports:
      - 27017:27017
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: example

  # Mongo Express
  mongo-express:
    image: mongo-express
    restart: always
    depends_on:
      - mongo
      - certbot
      - reverseproxy
```

```
    expose:
      - "8081"
    ports:
      - 8081:8081
    environment:
      ME_CONFIG_MONGODB_ADMINUSERNAME: root
      ME_CONFIG_MONGODB_ADMINPASSWORD: example
      VIRTUAL_HOST: mongo.mqttsec.microfast.ch
      LETSENCRYPT_HOST: mongo.mqttsec.microfast.ch
      LETSENCRYPT_EMAIL: cnvonallmen@hotmail.com

volumes:
  portainer_data:
  reverseproxy_data:
```

Listing A.1: Docker compose file used to run the software on the test system

A.2. Smoker configuration file

```
{
  "MqttSec": {
    "Broker": {
      "Tcp": {
        "Ssl": false,
        "Port": 1883,
        "PortSsl": 8883
      },
      "Websocket": {
        "Enabled": true,
        "Port": 8000,
        "Path": "/mqtt"
      },
      "Database": {
        "ClaimsCollectionName": "Claims",
        "ConnectionString": "mongodb://root:example@mongo:27017",
        "DatabaseName": "mqttsec"
      }
    }
  },
  "Logging": {
    "LogLevel": {
      "Default": "Error",
      "System": "Error",
      "Microsoft": "Error"
    },
    "Console": {
      "IncludeScopes": true
    }
  }
}
```

Listing A.2: Broker configuration file

B. Project management

B.1. Gitlab tasks

Task ID	Title	Milestone
1	Kick-Off	Project setup
4	Inital planning	Project setup
3	Tutor meeting	Finialize white paper authentication
5	Draw figures for successful / unsuccessful session establishments	Finialize white paper authentication
9	Finalize papper of client managed authentication	Finialize white paper authentication
8	Describe theoretical sequence of client managed authroization	Create white paper authorization
6	Implement crypto class library to generate client ID and NIZPK Proof	Implementation
7	Make MQTTNet compatible with enhanced authentication	Implementation
11	Implement authorization for MQTTNet Broker	Implementation
12	Implement izkp auth to mqttjs client	Implementation
13	Improve validation and logging	Implementation
14	Implement MongoDb Restriction store	Implementation
17	Verfiy restrictions by client signature	Implementation
18	Extend authorization logic	Implementation
19	Implement update and delete of claims	Implementation
15	Document implementation of authentication	Finialize documentation
16	Document broker evaluation	Finialize documentation
20	Document implementation of authorization	Finialize documentation
21	Update Prelimenaries	Finialize documentation
22	Finish Abstract / Management summary	Finialize documentation
23	Chapter Testing	Finialize documentation
24	Trust setting	Finialize documentation
26	Management Chapter	Finialize documentation
27	Document Usecases	Finialize documentation
28	Userguide how to use the enhanced authentication	Finialize documentation
29	Conclusion	Finialize documentation
33	Introduction	Finialize documentation
35	Authorization theory and impl document refactoring	Finialize documentation
36	Draw tree figure to illustrate restricted area	Finialize documentation
40	Create discussion chapter before conclusion	Finialize documentation
42	Add mqttjs documentation	Finialize documentation
43	Add future work chapter	Finialize documentation

Task ID	Title	Milestone
30	Create book entry	Create Poster / Film / Book entry
31	Create Poster	Create Poster / Film / Book entry
32	Create Film	Create Poster / Film / Book entry
37	Describe the use of digital signature in authn whitepaper	Reserve
39	General document refactoring	Reserve