Media Over QUIC
Internet-Draft
Intended status: Standards Track
Expires: 3 September 2026

S. Nandakumar
C. Jennings
Cisco
T. Meunier
Cloudflare Inc.

2 March 2026

Privacy Pass Authentication for Media over QUIC (MoQ)
draft-ietf-moq-privacy-pass-auth-02

Abstract

This document specifies the use of Privacy Pass architecture and issuance protocols for authorization in Media over QUIC (MoQ) transport protocol. It defines how Privacy Pass tokens can be integrated with MoQ's authorization framework to provide privacy-preserving authentication for subscriptions, fetches, publications, and relay operations while supporting fine-grained access control through prefix-based track namespace and track name matching rules.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://moq-wg.github.io/privacy-pass/draft-ietf-moq-privacy-pass-auth.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-moq-privacy-pass-auth/.

Discussion of this document takes place on the Media Over QUIC Working Group mailing list (mailto:moq@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/moq/. Subscribe at https://www.ietf.org/mailman/listinfo/moq/. Working Group information can be found at https://datatracker.ietf.org/wg/moq/.

Source for this draft and an issue tracker can be found at https://github.com/moq-wg/privacy-pass.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Media over QUIC (MoQ) [MoQ-TRANSPORT] provides a transport protocol for live and on-demand media delivery, real-time communication, and interactive content distribution over QUIC connections. The protocol supports a wide range of applications including video streaming, video conferencing, gaming, interactive broadcasts, and other latency-sensitive use cases. MoQ includes mechanisms for authorization through tokens that can be used to control access to media streams, interactive sessions, and relay operations.

Traditional authorization mechanisms often lack the privacy protection needed for modern media distribution scenarios, where users' viewing patterns and content preferences should remain private while still enabling fine-grained access control, namespace restrictions, and operational constraints.

Privacy Pass [RFC9576] provides a privacy-preserving authorization architecture that enables anonymous authentication through unlinkable tokens. The Privacy Pass architecture consists of four entities: Client, Origin, Issuer, and Attester, which work together to provide token-based authorization without compromising user privacy. The issuance protocols [RFC9578] define how these tokens are created and verified.

This document defines how Privacy Pass tokens can be integrated with MoQ's authorization framework to provide comprehensive access control for media streaming, real-time communication, and interactive content services while preserving user privacy through unlinkable authentication tokens.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Privacy Pass Architecture for MoQ

Privacy Pass Terminology defined in Section 2 of [RFC9576] is reused here. The Privacy Pass MoQ integration involves the following entities and their interactions:

* *Client*: The MoQ client requesting authorization to subscribe to, fetch, or publish media content. The client is responsible for obtaining Privacy Pass tokens through the attestation and issuance process, and presenting these tokens when requesting MoQ operations such as SUBSCRIBE, FETCH, PUBLISH, or PUBLISH_NAMESPACE.

* *MoQ Relay*: The MoQ relay server that forwards media content and verifies that clients are authorized. The relay validates Privacy Pass tokens presented by clients, enforces access policies, and forwards authorized requests to other relays. Relays maintain configuration for trusted issuers and validate token signatures and metadata.

* *Privacy Pass Issuer*: The entity that issues Privacy Pass tokens to clients after successful attestation. The issuer operates the token issuance protocol, manages cryptographic keys. The issuer creates tokens with appropriate MoQ-specific metadata.

* *Privacy Pass Attester*: The entity that attests to properties of clients for the purposes of token issuance. The attester verifies client credentials, subscription status, or other eligibility criteria. Common attestation methods include username/password, OAuth, device certificates, or other authentication mechanisms.

2.1. Joint Attester and Issuer

In the below deployment, the MoQ relay and Privacy Pass issuer are operated by different entities to enhance privacy through separation of concerns. This corresponds to Section 4.4 of [RFC9576].

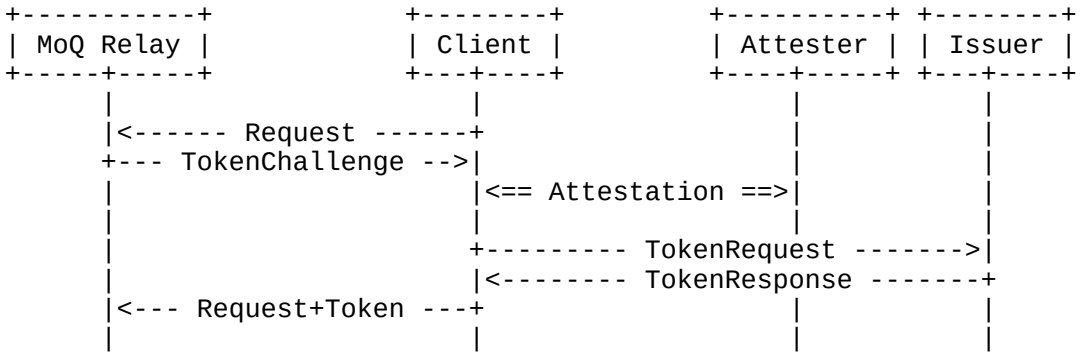


Figure 1: Separated Issuer and Relay Architecture

In certain deployments the MoQ relay and Privacy Pass issuer may be operated by the same entity to simplify key management and policy coordination. This is the Privacy Pass deployment described in Section 4.2 of [RFC9576].

2.2. Shared Origin, Attester, Issuer with a Reverse Flow

The flow described above can be used to bootstrap a shared origin-attester-issuer flow, as described in Section 4.2 of [RFC9576]. The MoQ relay plays all roles (origin, attester, and issuer), allowing it to use privately verifiable token types registered in [PRIVACYPASS-IANA].

In this scenario, the MoQ relay origin would accept tokens signed by two issuers:

1. Type 0x0002 token signed by the bootstrap issuer from Section 2.1

2. Type 0x0001, 0x0005, or 0xE5AC tokens signed by its own issuer.

This two-phase approach provides several advantages:

* *Bootstrapping*: The initial publicly verifiable token (0x0002) establishes trust without requiring the relay to share private keys with external verifiers.

* *Efficiency*: Subsequent privately verifiable tokens allow batched issuance, amortizing cryptographic costs across multiple operations.

* *Privacy*: Each token presentation is unlinkable, even when obtained from the same credential.

2.2.1. Reverse Flow Overview

The reverse flow, as described in Section 4 of [PRIVACYPASS-REVERSE-FLOW], allows a client to exchange a publicly verifiable token for privately verifiable tokens (or credentials) issued directly by the MoQ relay.

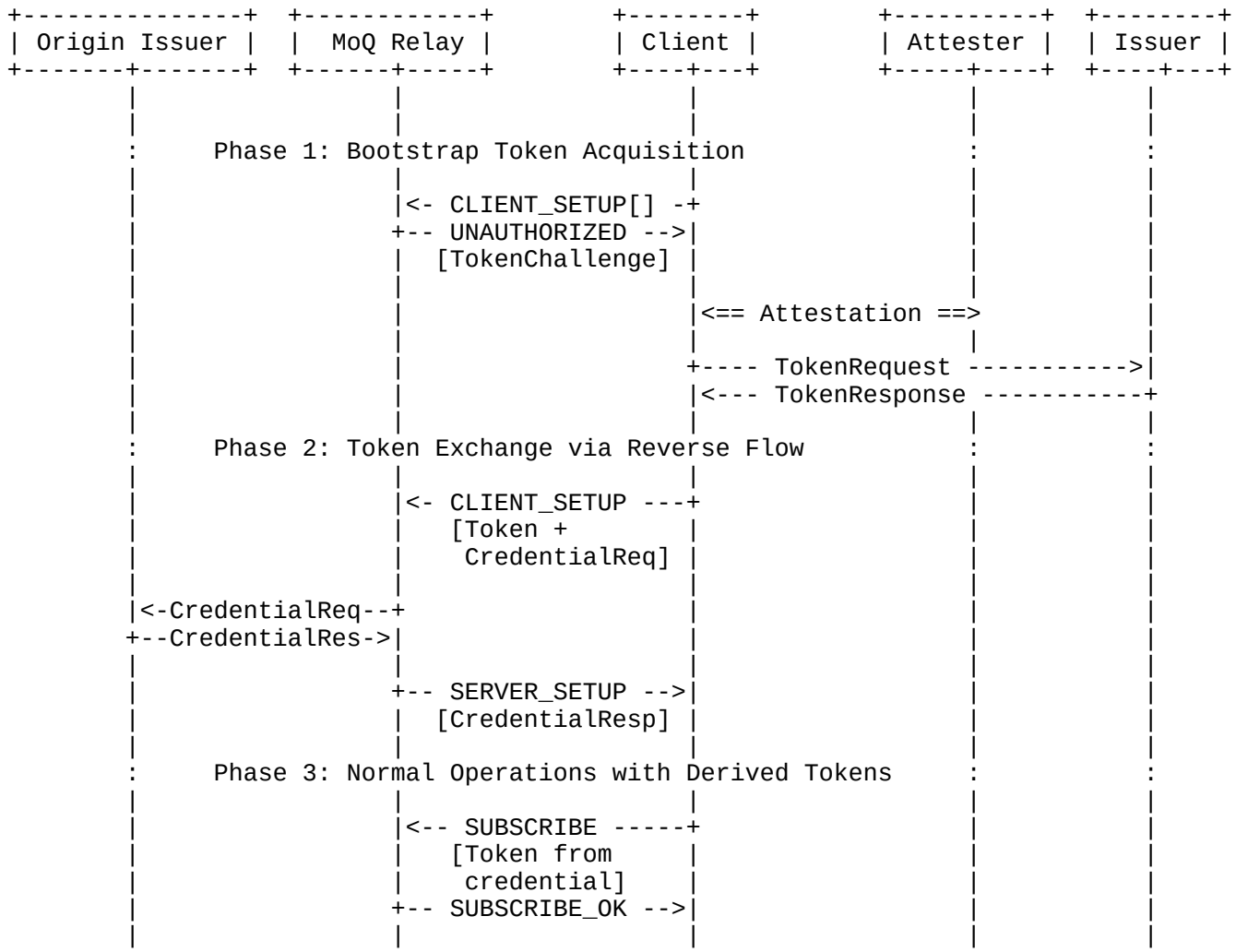


Figure 2: Complete Reverse Flow Authorization

2.2.2. Detailed Reverse Flow Steps

Phase 1: Bootstrap Token Acquisition

1. The client initiates a connection with CLIENT_SETUP without authorization.
2. The MoQ relay responds with UNAUTHORIZED containing a TokenChallenge specifying a publicly verifiable token type (0x0002).
3. The client performs attestation with an external attester/issuer.
4. The client obtains a publicly verifiable token.

Phase 2: Token Exchange via Reverse Flow

1. The client sends CLIENT_SETUP with:
    * The publicly verifiable Token from Phase 1
    * A CredentialRequest (or TokenRequest) for a privately verifiable token type (0x0001, 0x0005, or 0xE5AC)
2. The MoQ relay validates the bootstrap token, then processes the credential request using its internal issuer.
3. The MoQ relay responds with SERVER_SETUP containing:
    * A CredentialResponse (or TokenResponse) with the privately verifiable credential/tokens

Phase 3: Normal Operations

1. For subsequent operations (SUBSCRIBE, PUBLISH, FETCH), the client presents tokens derived from the credential obtained in Phase 2.
2. The MoQ relay validates tokens locally using its private verification key.

2.2.3. Credential Request/Response Encoding

When using the reverse flow, the GenericBatchTokenRequest in ClientPrivateTokenAuth contains the credential or token request for the privately verifiable token type:

* For 0x0001 or 0x0005: TokenRequest as defined in Section 5.1 of [PRIVACYPASS-BATCHED]

* For 0xE5AC: CredentialRequest as defined in Section 7.1 of [PRIVACYPASS-ARC]

Similarly, GenericBatchTokenResponse in ServerPrivateTokenAuth contains:

* For 0x0001 or 0x0005: TokenResponse as defined in Section 5.2 of [PRIVACYPASS-BATCHED]

* For 0xE5AC: CredentialResponse as defined in Section 7.2 of [PRIVACYPASS-ARC]

2.3. Trust Model

The architecture assumes the following trust relationships based on Section 3 of [RFC9576]:

* Relays trust issuers to properly validate client eligibility before issuing tokens

* Issuers trust attesters to accurately verify client eligibility

3. Privacy Pass Token Integration

This section describes how Privacy Pass tokens are integrated into the MoQ transport protocol to provide privacy-preserving authorization for various media operations.

3.1. Token Types for MoQ Authorization

This specification uses the below existing Privacy Pass token types:

Publicly verifiable token types

* 0x0002 (Blind RSA (2048-bit)): Defined in Section 6 of [RFC9578]. Uses blind RSA signatures ([RFC9474]) for deployments requiring distributed validation across multiple relays.

Privately verifiable token types

* 0x0001 (VOPRF(P-384, SHA-384)): Defined in Section 6 of [RFC9578]. Uses VOPRF ([RFC9497]) for deployments where the origin is the issuer. Issuance can be batched as defined in Section 5 of [PRIVACYPASS-BATCHED].

* 0x0005 (VOPRF(ristretto255, SHA-512)): Defined in Section 8.1 of [PRIVACYPASS-BATCHED]. Uses VOPRF ([RFC9497]) for deployments where the origin is the issuer. Issuance can be batched as defined in Section 5 of [PRIVACYPASS-BATCHED].

* 0xE5AC (ARC(P-256)): Anonymous Rate Limit Credentials Token using [ARC]. Tokens are presented by clients based on an issued credential and up to a presentation_limit.

3.2. Token Structure

Privacy Pass tokens used in MoQ MUST follow the structure defined in Section 2.2 of [RFC9577] for the PrivateToken HTTP authentication scheme. The token structure includes:

* *Token Type*: 2-byte identifier specifying the issuance protocol used

* *Nonce*: 32-byte client-generated random value for uniqueness

* *Challenge Digest*: 32-byte SHA-256 hash of the TokenChallenge

* *Token Key ID*: Variable-length identifier for the issuer’s public key

* *Authenticator*: Variable-length cryptographic proof bound to the token

3.2.1. Token Challenge Structure for MoQ

MoQ-specific TokenChallenge structures use the default format defined in Section 2.1 of [RFC9577] with MoQ-specific parameters in the origin_info field, reproduced thereafter for convenience:

```
struct {
    uint16_t token_type;
    opaque issuer_name<1..2^16-1>;
    opaque redemption_context<0..32>;
    opaque origin_info<0..2^16-1>;
} TokenChallenge;
```

For MoQ usage, authorization scope information can be encoded by the origin within origin_info field. This is encoded in the Token at issuance time when types 0x0001, 0x0002, 0x0005 are used. When clients present a credential such as with [ARC], the scope may be restricted at presentation time.

Origins MAY use redemption_context to scope token use to properties of the client session. As described in Section 2.1.1.2 of [RFC9577], redemption context can be set to 32-byte random nonce, to the hash of a specific time window, or even derived from the client's ASN.

3.2.2. MoQ Actions

MoQ operations are identified by the following action values, aligned with MoQTransport control message types:

Action	Value	Reference
CLIENT_SETUP	0	Section 9.3 of [MoQ-TRANSPORT]
SERVER_SETUP	1	Section 9.3 of [MoQ-TRANSPORT]
PUBLISH_NAMESPACE	2	Section 9.20 of [MoQ-TRANSPORT]
SUBSCRIBE_NAMESPACE	3	Section 9.25 of [MoQ-TRANSPORT]
SUBSCRIBE	4	Section 9.9 of [MoQ-TRANSPORT]
REQUEST_UPDATE	5	Section 9.11 of [MoQ-TRANSPORT]
PUBLISH	6	Section 9.13 of [MoQ-TRANSPORT]
FETCH	7	Section 9.16 of [MoQ-TRANSPORT]
TRACK_STATUS	8	Section 9.19 of [MoQ-TRANSPORT]

Table 1: MoQ Action Values

The default authorization policy is "blocked" - all actions are denied unless explicitly permitted by a token scope.

MoQAction wire representation is as follows

```
enum {
    CLIENT_SETUP(0),
    SERVER_SETUP(1),
    PUBLISH_NAMESPACE(2),
    SUBSCRIBE_NAMESPACE(3),
    SUBSCRIBE(4),
    REQUEST_UPDATE(5),
    PUBLISH(6),
    FETCH(7),
    TRACK_STATUS(8),
    (255)
} MoQAction;
```

3.2.3. Match Types

Match rules for namespaces and track names support the following types:

Match Type	Value	Description
MATCH_EXACT	0	Value must equal the pattern exactly
MATCH_PREFIX	1	Value must start with the pattern
MATCH_SUFFIX	2	Value must end with the pattern
MATCH_CONTAINS	3	Value must contain the pattern as substring

Table 2: Match Type Values

Track namespaces in MoQ are represented as ordered tuples of byte strings (e.g., ["example.com", "live", "sports"]). Match rules operate on these tuples at tuple element boundaries. The pattern in a MatchRule (defined in Section 3.2.4) is also a tuple of byte strings, and matching is performed element-by-element.

As for track names, match rules can be applied directly given there is a single tuple element.

No normalization is performed on namespace tuple elements or track name values before matching. Comparisons are performed as byte-level operations on each tuple element.

MatchType wire representation is as follows

```
enum {
    MATCH_EXACT(0),
    MATCH_PREFIX(1),
    MATCH_SUFFIX(2),
    MATCH_CONTAINS(3),
    (255)
} MatchType;
```

3.2.4. Authorization Scope Structure (origin_info)

When authorization scope is bound at issuance time, the origin_info field contains a binary-encoded MoQAuthorizationInfo structure:

```
struct {
    opaque element<0..2^16-1>;
} TupleElement;

struct {
    TupleElement elements<0..2^16-1>;
} NamespaceTuple;

struct {
    MatchType match_type;
    NamespaceTuple value;
} NamespaceMatchRule;

struct {
    MatchType match_type;
    opaque value<0..2^16-1>;
} TrackNameMatchRule;

struct {
    MoQAction actions<1..2^8-1>;
    NamespaceMatchRule namespace_match;
    TrackNameMatchRule track_name_match;
} MoQAuthScope;

struct {
    MoQAuthScope scopes<1..2^8-1>;
} MoQAuthorizationInfo;
```

A token MAY contain multiple MoQAuthScope entries to authorize different combinations of actions and resource patterns. Authorization succeeds if ANY scope in the token permits the requested operation.

3.2.5. Examples

The following examples illustrate authorization scope configurations:

Subscribe to live sports namespace (prefix match):

```
MoQAuthScope {
    actions = [SUBSCRIBE(4)],
    namespace_match = {
        match_type = MATCH_PREFIX(1),
        value = ["sports.example.com", "live"]
    },
    track_name_match = {
        match_type = MATCH_PREFIX(1),
        value = ""
    }
}
```

This matches namespace tuples like ["sports.example.com", "live", "soccer"] and ["sports.example.com", "live", "tennis", "finals"].

Publish to specific meeting track (exact match):

```
MoQAuthScope {
    actions = [PUBLISH(6)],
    namespace_match = {
        match_type = MATCH_EXACT(0),
        value = ["meetings.example.com", "meeting", "m123"]
    },
    track_name_match = {
        match_type = MATCH_PREFIX(1),
        value = "audio-"
    }
}
```

This matches only the exact namespace tuple ["meetings.example.com", "meeting", "m123"] with track names starting with "audio-".

Fetch video-on-demand with suffix matching:

```
MoQAuthScope {
    actions = [FETCH(7)],
    namespace_match = {
        match_type = MATCH_CONTAINS(3),
        value = ["vod", "movies"]
    },
    track_name_match = {
        match_type = MATCH_SUFFIX(2),
        value = ".mp4"
    }
}
```

This matches namespace tuples containing the contiguous subsequence ["vod", "movies"], such as ["example.com", "vod", "movies", "action"].

3.3. Track Namespace and Track Name Matching Rules

This specification defines matching rules for track namespaces and track names to enable fine-grained access control while maintaining privacy. Both namespace and track name matching use the same MatchRule structure and algorithm.

3.3.1. Match Rule Evaluation

Given a MatchRule and a target value (namespace tuple or track name), the match succeeds according to the following rules. For namespace matching, both the pattern and target are tuples of byte strings; matching operates at tuple element boundaries.

MATCH_EXACT (0):

The target MUST be identical to the pattern. For namespace tuples, this means the same number of elements with each element byte-for-byte identical. The pattern tuple ["example.com", "live"] matches only ["example.com", "live"], not ["example.com", "live", "sports"].

MATCH_PREFIX (1):

The target MUST start with the pattern at tuple element boundaries. The pattern tuple ["example.com", "live"] matches ["example.com", "live", "sports"] and ["example.com", "live", "news", "breaking"] but not ["example.com", "vod"]. Note that ["example.com", "liv"] does NOT match ["example.com", "live"] since matching is at element boundaries.

MATCH_SUFFIX (2):

The target MUST end with the pattern at tuple element boundaries. The pattern tuple ["audio"] matches ["meeting123", "audio"] and ["conference", "room1", "audio"] but not ["audio", "opus"].

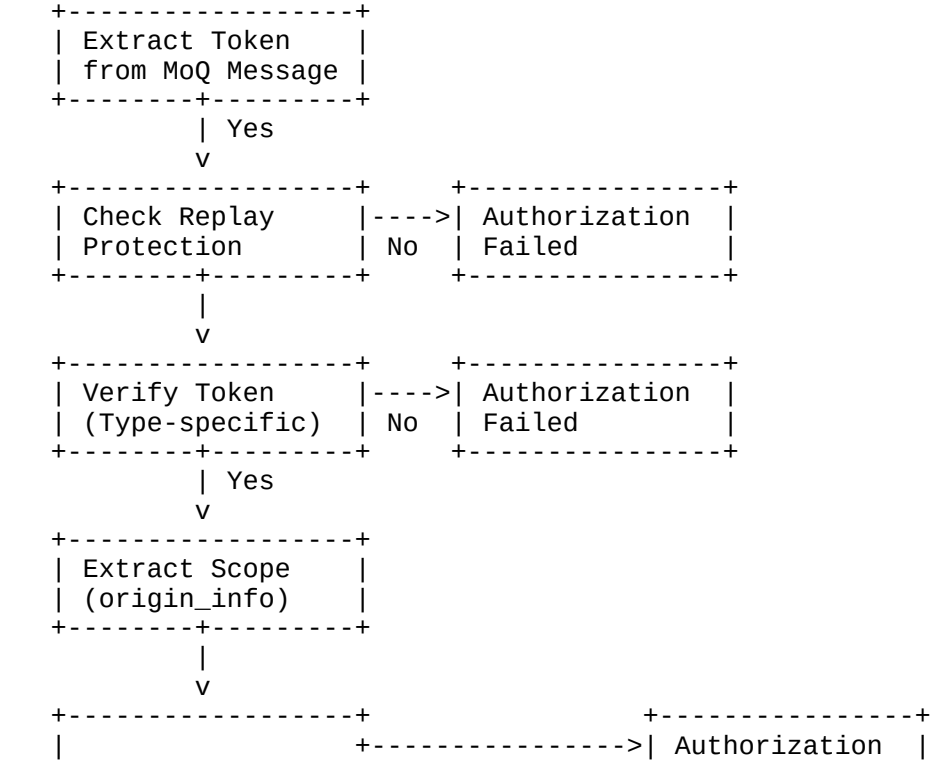
MATCH_CONTAINS (3):

The target MUST contain the pattern as a contiguous subsequence of tuple elements. The pattern tuple ["live", "sports"] matches ["example.com", "live", "sports", "soccer"] but the single-element pattern ["sports"] does NOT match ["live-sports", "channel"] since "sports" is a substring within an element, not a complete element.

Note: To match all values, use MATCH_PREFIX with an empty pattern ([] for namespaces or "" for track names). An empty pattern is a prefix of every value.

3.3.2. Matching Algorithm

When a MoQ relay receives a request with a Privacy Pass token, it performs the following validation steps to determine whether to authorize the requested operation:



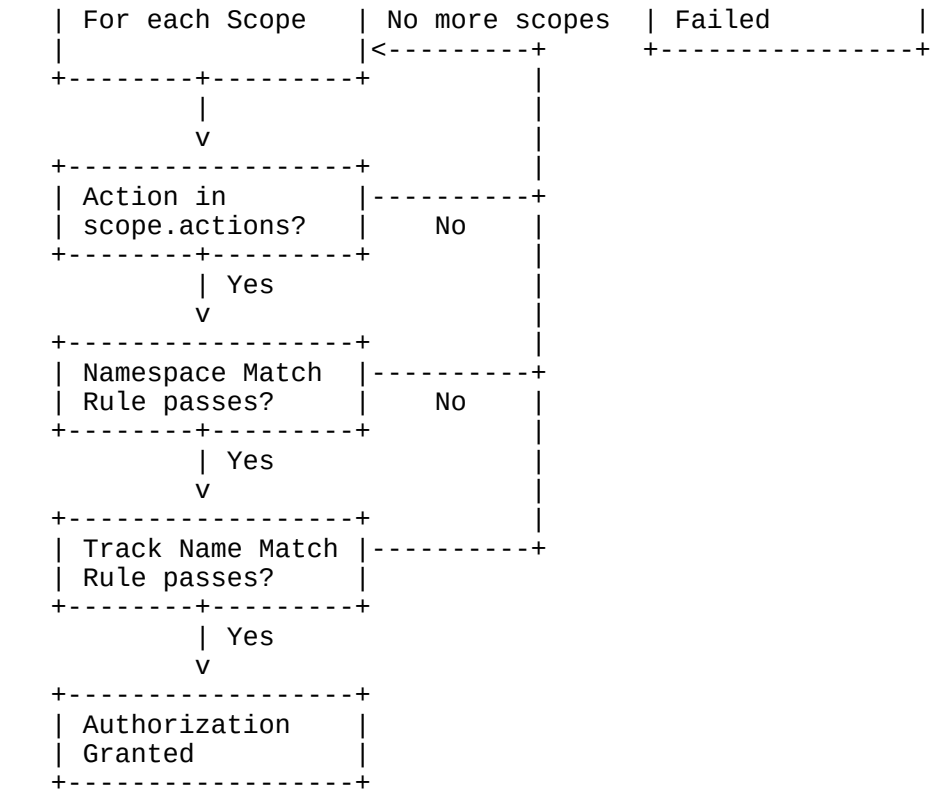


Figure 3: Token Validation and Matching Algorithm

1. *Token Extraction*: Extract the Privacy Pass token from the MoQ control message (SETUP, SUBSCRIBE, FETCH, PUBLISH, PUBLISH_NAMESPACE, or other operation).

2. *Token Verification*: Verify the token using the appropriate method for the token type:

   * Token Type 0x0001 or 0x0005 (VOPRF): Verify using the issuer's private validation key

   * Token Type 0x0002 (Blind RSA): Verify using the issuer's public verification key

   * Token Type 0xE5AC (ARC): Verify the presentation proof using the issuer's public parameters

3. *Replay Protection*: Validate that the token has not been replayed:

* Check token nonce uniqueness within the configured replay window

* Verify token expiration timestamp if present in token metadata

4. *Scope Extraction*: Extract authorization scope from the token:

   * If using origin_info: Decode the MoQAuthorizationInfo structure

5. *Scope Evaluation*: For each MoQAuthScope in the token, check if the requested operation is authorized:

   a. *Action Check*: Verify the requested MoQ action (from Section 3.2.2) is present in the scope’s actions list

   b. *Namespace Match*: Apply the namespace_match rule to the requested track namespace using the algorithm in Section 3.2.3

   c. *Track Name Match*: Apply the track_name_match rule to the requested track name using the algorithm in Section 3.2.3

   d. If all three checks pass, authorization succeeds for this scope

6. *Authorization Decision*: Access is granted if and only if:

   * Token verification succeeds (step 2)

   * Replay protection passes (step 3)

   * At least one scope in the token authorizes the operation (step 5)

If authorization fails, an error is returned as specified in Section 3.4.5.

3.4. Token in MOQ Messages

Privacy Pass tokens are provided to MoQ relays using the existing MoQ authorization framework with the following adaptations:

3.4.1. SETUP Message Authorization

For connection-level authorization, Privacy Pass tokens are included in the SETUP message’s authorization parameter (Section 9.3.1.5 of [MoQ-TRANSPORT]).

```
SETUP {
    Version = 1,
    Parameters = [
        {
            Type = AUTHORIZATION,
            Value = PrivateTokenAuth
        }
    ]
}

type PrivateTokenAuth = ClientPrivateTokenAuth | ServerPrivateTokenAuth;
```

For CLIENT_SETUP, the authorization value uses GenericBatchTokenRequest as defined in Section 6.1 of [PRIVACYPASS-BATCHED] as follows:

```
struct {
    uint8_t auth_scheme = 0x01;
    Token token;
    GenericBatchTokenRequest token_requests;
} ClientPrivateTokenAuth;
```

For SERVER_SETUP, the authorization value uses GenericBatchTokenResponse as defined in Section 6.2 of [PRIVACYPASS-BATCHED] as follows:

```
struct {
    uint8_t auth_scheme = 0x01;
    Token token;
    GenericBatchTokenResponse token_responses;
} ServerPrivateTokenAuth;
```

When batch issuance is not used, token_requests and token_responses are empty (length = 0).

The Token structure is prepended by a two-byte token type identifier as registered with IANA:

```
struct {
   uint16_t token_type; /* From the IANA Privacy Pass Token Types Registry */
   select (token_type) { /* Rest of the token */
     case (0x0001, 0x0002, 0x0005):
         uint8_t nonce[32];
         uint8_t challenge_digest[32];
         uint8_t token_key_id[Nid];
         uint8_t authenticator[Nk];
     case (other): /* Other token types from the IANA Privacy Pass Token Types Registry *
/
         opaque remainder<0..2^16-1>;
   }
} Token;
```

Where Nk is determined by token_type per the [PRIVACYPASS-IANA].

Unknown token types MUST be rejected.

3.4.2. MoQ Operation-Level Authorization

For individual MoQ operation authorization, tokens are included in operation-specific control messages:

```
SUBSCRIBE {
    Track_Namespace = "sports.example.com/live/soccer",
    Track_Name = "video",
    Parameters = [
        {
            Type = AUTHORIZATION,
            Value = PrivateTokenAuth
        }
    ]
}
```

3.4.3. Continuous Authorization with Batched Tokens

Long-lived MoQ sessions (such as live streaming or real-time communication) require periodic re-authorization to ensure continued eligibility. Unlike JWT-based approaches that use explicit revalidation intervals, Privacy Pass can achieve continuous authorization through batched token issuance.

During the initial SETUP exchange, clients can request multiple tokens via GenericBatchTokenRequest (defined in Section 6.1 of [PRIVACYPASS-BATCHED]). Each token in the batch is independently valid and can be presented for subsequent operations or periodic re-authorization.

Batched Token Usage Timeline:

```
Time 0:     CLIENT_SETUP with Token_1, request batch of N tokens
            SERVER_SETUP with batch of N tokens

Time T:     SUBSCRIBE with Token_2 (from batch)

Time 2T:    Client presents Token_3 for continued authorization
            (proactive re-auth before relay requests it)

Time 3T:    Relay requests re-authorization
            Client presents Token_4
```

Relays MAY request periodic re-authorization by sending a TokenChallenge in a REQUEST_ERROR message. Clients SHOULD present a fresh token from their batch in response if any satisfy the new TokenChallenge. If not, they SHOULD perform a new issuance process.

When using [ARC] tokens (0xE5AC), the credential's presentation_limit controls how many times the client can present tokens from a single credential issuance. This provides rate limiting while preserving unlinkability between presentations.

Deployment Considerations:

* Batch size SHOULD be sufficient for the expected session duration
* Relays SHOULD configure re-authorization intervals based on content sensitivity and trust requirements
* Clients SHOULD request new token batches before exhausting their supply
* For high-security deployments, shorter re-authorization intervals with smaller batches provide stronger revocation guarantees

3.4.4. Continuous Authorization with Reverse Flow

If the client and the relay support it, a Relay MAY perform continuous authentication using a reverse flow.

To do so, when presenting PrivateTokenAuth, a client MUST send at least one GenericBatchTokenRequest. The Relay then acts as a reverse issuer, and issues the corresponding number of GenericBatchTokenResponse.

Tokens obtained this way can be presented by the Client to maintain the continuity of the session without linkability.

Reverse Flow Token Usage Timeline:

```
Time 0:     CLIENT_SETUP with Token_1, request batch of 1 token
            SERVER_SETUP with batch of 1 token

Time T:     SUBSCRIBE with Token_2 (from batch), request batch of 1 token
            Relay responds with batch of 1 token

Time 2T:    Client presents Token_3 (from batch of time T), request batch of 1 token
            Relay responds with batch of 1 token
```

3.4.5. Errors

If the authentication fails for any reason, the server MUST send an error. The error response includes a TokenChallenge to enable the client to obtain a valid token and retry the operation.

3.4.5.1. SETUP Errors

If authentication fails during SETUP, the Relay MUST terminate the connection with the UNAUTHORIZED (0x02) Termination Error Code defined in Section 3.4 of [MoQ-TRANSPORT]. The termination reason phrase MUST contain a MoQAuthChallenge structure:

```
struct {
    TokenChallenge challenges<1..2^16-1>;
} MoQAuthChallenge;
```

The challenges field lists token challenges the relay accepts, ordered by preference (most preferred first). This allows clients to select an appropriate issuance protocol based on supported token types and issuers. Each challenge specifies a token type, issuer, and optional scope, enabling relays to accept different issuers or scopes for different token types. Relay MUST include at least one challenge.

3.4.5.2. Operation Errors

If the error occurs over an established connection, the Relay MUST send a REQUEST_ERROR defined in Section 9.8 of [MoQ-TRANSPORT].

The error code MUST be one of:

Error Code	Name	Description
0x0100	TOKEN_MISSING	No token provided when required
0x0101	TOKEN_INVALID	Token signature verification failed
0x0102	TOKEN_EXPIRED	Token has expired or been revoked
0x0103	TOKEN_REPLAYED	Token nonce has been seen before
0x0104	SCOPE_MISMATCH	Token scope does not authorize this operation
0x0105	ISSUER_UNKNOWN	Token issuer is not trusted by this relay
0x0106	TOKEN_MALFORMED	Token cannot be parsed correctly

Table 3: Privacy Pass Authorization Error Codes

The reason phrase in REQUEST_ERROR MUST contain a MoQAuthChallenge structure when the client should retry with a new token, encoded as a byte-string.

3.4.5.3. TokenChallenge Construction

Each TokenChallenge in MoQAuthChallenge MUST be constructed as follows:

* token_type: The token type for this challenge
* issuer_name: The issuer name that can issue tokens for this challenge
* redemption_context: A fresh 32-byte random value, or empty if the relay accepts tokens with any redemption context
* origin_info: The relay’s origin identifier, optionally including the required authorization scope

Different challenges MAY specify different issuers or scopes for different token types. When origin_info is empty, the relay accepts tokens with any scope and performs authorization based solely on the token's embedded scope information.

3.4.5.4. Error Response Example

```
REQUEST_ERROR {
    Request_ID = 42,
    Error_Code = 0x0104,  /* SCOPE_MISMATCH */
    Reason = MoQAuthChallenge {
        challenges = [
            TokenChallenge {
                token_type = 0x0002,
                issuer_name = "public-issuer.example.com",
                redemption_context = <32 random bytes>,
                origin_info = <authorization scope>
            },
            TokenChallenge {
                token_type = 0xE5AC,
                issuer_name = "relay.example.com",
                redemption_context = <32 random bytes>,
                origin_info = <authorization scope>
            },
            TokenChallenge {
                token_type = 0x0001,
                issuer_name = "relay.example.com",
                redemption_context = <32 random bytes>,
                origin_info = <authorization scope>
            }
        ]
    }
}
```

3.4.5.5. Control Message Authorization Failures

When authorization fails for MoQ control messages other than SETUP, the relay returns a REQUEST_ERROR with the appropriate error code from Table 3. The client MAY retry the operation with a valid token obtained using the TokenChallenge from the error response.

As per Section 3.4.4 of [MoQ-TRANSPORT], implementations MAY elevate request-specific errors to session-level errors. This elevation is appropriate when:

* The authorization failure indicates a systemic issue (e.g., all client tokens are from an untrusted issuer)

* Continuing the session would be futile due to policy restrictions

* The error represents a security concern requiring session termination

Implementations need to consider the impact on other outstanding subscriptions before elevating to session-level errors.

4. Example Authorization Flow

Below shows an example deployment scenario where the relay has been configured with the necessary validation keys and content policies. The relay can verify Privacy Pass tokens locally and deliver media directly without contacting the Issuer. This example uses publicly verifiable tokens.

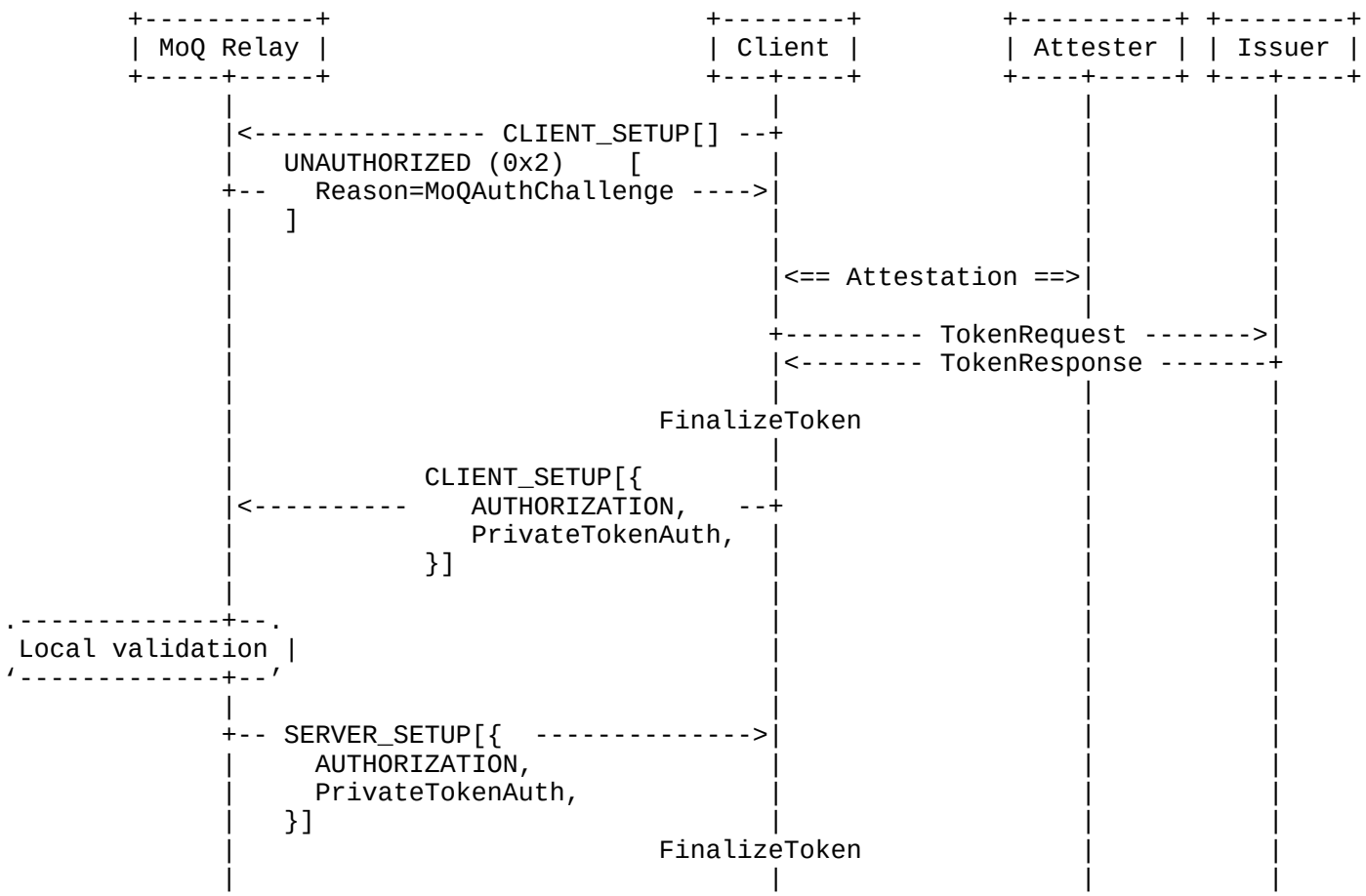


Figure 4: Direct Relay Authorization Flow

The MoQAuthChallenge in the UNAUTHORIZED response contains:

* A TokenChallenge with the relay's issuer configuration

* A list of supported_token_types (e.g., [0x0002, 0xE5AC])

This allows the client to select the appropriate issuance protocol based on its capabilities and the available attesters/issuers.

5. Security Considerations

TODO: Add considerations for the security and privacy of the Privacy Pass tokens.

* Token Replay

* Token harvest

* Key rotation

* Use of TLS

6. IANA Considerations

6.1. MoQ Privacy Pass Auth Scheme Registry

IANA is requested to create a new registry titled "MoQ Privacy Pass Auth Schemes" with the following initial contents:

Value	Name	Reference
0x00	Reserved	This document
0x01	PrivateTokenAuth	This document

Table 4: MoQ Privacy Pass Auth Schemes

New entries in this registry require Specification Required registration policy.

6.2. MoQ Action Registry

IANA is requested to create a new registry titled "MoQ Actions for Privacy Pass Authorization" with the following initial contents:

Value	Action	Reference
0	CLIENT_SETUP	Section 3.2.2
1	SERVER_SETUP	Section 3.2.2
2	PUBLISH_NAMESPACE	Section 3.2.2
3	SUBSCRIBE_NAMESPACE	Section 3.2.2
4	SUBSCRIBE	Section 3.2.2
5	REQUEST_UPDATE	Section 3.2.2
6	PUBLISH	Section 3.2.2
7	FETCH	Section 3.2.2
8	TRACK_STATUS	Section 3.2.2
9-254	Unassigned	
255	Reserved	This document

Table 5: MoQ Actions Registry

New entries in this registry require Specification Required registration policy. Values SHOULD align with MoQTransport control message types where applicable.

6.3. MoQ Match Type Registry

IANA is requested to create a new registry titled "MoQ Match Types for Privacy Pass Authorization" with the following initial contents:

Value	Match Type	Reference
0	MATCH_EXACT	Section 3.2.3
1	MATCH_PREFIX	Section 3.2.3
2	MATCH_SUFFIX	Section 3.2.3
3	MATCH_CONTAINS	Section 3.2.3
4-254	Unassigned	
255	Reserved	This document

Table 6: MoQ Match Types Registry

New entries in this registry require Specification Required registration policy.

6.4. MoQ Privacy Pass Error Code Registry

IANA is requested to create a new registry titled "MoQ Privacy Pass Authorization Error Codes" with the following initial contents:

Value	Name	Reference
0x0100	TOKEN_MISSING	Section 3.4.5
0x0101	TOKEN_INVALID	Section 3.4.5
0x0102	TOKEN_EXPIRED	Section 3.4.5
0x0103	TOKEN_REPLAYED	Section 3.4.5
0x0104	SCOPE_MISMATCH	Section 3.4.5
0x0105	ISSUER_UNKNOWN	Section 3.4.5
0x0106	TOKEN_MALFORMED	Section 3.4.5
0x0107-0x01FF	Unassigned	

Table 7: MoQ Privacy Pass Error Codes Registry

New entries in this registry require Specification Required registration policy. Values are allocated from the 0x0100-0x01FF range reserved for Privacy Pass authorization errors.

7. References

7.1. Normative References

[ARC] Yun, C., Wood, C. A., and A. F. Faz-Hernandez, "Anonymous Rate-Limited Credentials Cryptography", Work in Progress, Internet-Draft, draft-ietf-privacypass-arc-crypto-00, 3 February 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-arc-crypto-00>.

[MoQ-TRANSPORT] Nandakumar, S., Vasiliev, V., Swett, I., and A. Frindell, "Media over QUIC Transport", Work in Progress, Internet-Draft, draft-ietf-moq-transport-16, 13 January 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-moq-transport-16>.

[PRIVACYPASS-ARC] Yun, C., Wood, C. A., and A. F. Faz-Hernandez, "Privacy Pass Issuance Protocol for Anonymous Rate-Limited Credentials", Work in Progress, Internet-Draft, draft-ietf-privacypass-arc-protocol-00, 4 February 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-arc-protocol-00>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC9474] Denis, F., Jacobs, F., and C. A. Wood, "RSA Blind Signatures", RFC 9474, DOI 10.17487/RFC9474, October 2023, <https://www.rfc-editor.org/rfc/rfc9474>.

[RFC9497] Davidson, A., Faz-Hernandez, A., Sullivan, N., and C. A. Wood, "Oblivious Pseudorandom Functions (OPRFs) Using Prime-Order Groups", RFC 9497, DOI 10.17487/RFC9497, December 2023, <https://www.rfc-editor.org/rfc/rfc9497>.

[RFC9576] Davidson, A., Iyengar, J., and C. A. Wood, "The Privacy Pass Architecture", RFC 9576, DOI 10.17487/RFC9576, June 2024, <https://www.rfc-editor.org/rfc/rfc9576>.

[RFC9577] Pauly, T., Valdez, S., and C. A. Wood, "The Privacy Pass HTTP Authentication Scheme", RFC 9577, DOI 10.17487/RFC9577, June 2024, <https://www.rfc-editor.org/rfc/rfc9577>.

[RFC9578] Celi, S., Davidson, A., Valdez, S., and C. A. Wood, "Privacy Pass Issuance Protocols", RFC 9578, DOI 10.17487/RFC9578, June 2024, <https://www.rfc-editor.org/rfc/rfc9578>.

7.2. Informative References

[PRIVACYPASS-BATCHED]
Robert, R., Wood, C. A., and T. Meunier, "Batched Token Issuance Protocol", Work in Progress, Internet-Draft, draft-ietf-privacypass-batched-tokens-07, 25 February 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-batched-tokens-07>.

[PRIVACYPASS-IANA]
"Privacy Pass IANA", n.d., <https://www.iana.org/assignments/privacy-pass/privacy-pass.xhtml>.

[PRIVACYPASS-REVERSE-FLOW]
Meunier, T., "Privacy Pass Reverse Flow", Work in Progress, Internet-Draft, draft-meunier-privacypass-reverse-flow-03, 16 February 2026, <https://datatracker.ietf.org/doc/html/draft-meunier-privacypass-reverse-flow-03>.

[RFC9458] Thomson, M. and C. A. Wood, "Oblivious HTTP", RFC 9458, DOI 10.17487/RFC9458, January 2024, <https://www.rfc-editor.org/rfc/rfc9458>.

Appendix A. Acknowledgments

TODO acknowledge.

Appendix B. Change Log

RFC Editor’s Note: Please remove this section prior to publication of a final version of this document.

B.1. Since draft-ietf-moq-privacy-pass-auth-02

* Expanded reverse flow documentation with three-phase flow (bootstrap, exchange, operations)
* Defined MoQAuthChallenge structure for error responses with supported_token_types
* Added TokenChallenge construction requirements
* Added control message authorization failure handling section
* Documented credential request/response encoding for different token types

B.2. Since draft-ietf-moq-privacy-pass-auth-01

* Replace text-based moq-scope with binary TLS presentation language structures
* Add MoQ Actions registry aligned with MoQTransport control message types
* Add Match Types registry with exact, prefix, suffix, and contains matching
* Define MoQAuthorizationInfo structure for origin_info encoding
* Add continuous authorization section using reverse flow
* Add continuous authorization section using batched tokens
* Add IANA registries for auth schemes, actions, and match types
* Define error handling
* Integrate privacy pass reverse flow within PrivateTokenAuth
* MoQ definition now follow draft-ietf-moq-transport-16
* Update dependencies
* Removed b64 encoding given MoQ can use bytes directly

B.3. Since draft-ietf-moq-privacy-pass-auth-00

* Add Thibault Meunier as Coauthor

* Add support for Reverse flow to be deploy and scale friendly way to get tokens

Authors' Addresses

Suhas Nandakumar
Cisco
Email: snandaku@cisco.com

Cullen Jennings
Cisco
Email: fluffy@iii.ca

Thibault Meunier
Cloudflare Inc.
Email: ot-ietf@thibault.uk