Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 1 September 2025

C. Jennings
S. Nandakumar
R. Barnes
Cisco

28 February 2025

End-to-End Secure Objects for Media over QUIC Transport
draft-jennings-moq-secure-objects-02

Abstract

This document describes an end-to-end authenticated encryption scheme for application objects intended to be delivered over Media over QUIC Transport (MOQT). We reuse the SFrame scheme for authenticated encryption of media objects, while suppressing data that would be redundant between SFrame and MOQT, for an efficient on-the-wire representation.

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://suhashere.github.io/moq-secure-objects/#go.draft-jennings-moq-secure-objects.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-jennings-moq-secure-objects/.

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/.

Source for this draft and an issue tracker can be found at https://github.com/suhasHere/moq-secure-objects.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Copyright Notice

Table of Contents

1. Introduction

Media Over QUIC Transport (MOQT) is a protocol that is optimized for the QUIC protocol, either directly or via WebTransport, for the dissemination of delivery of low latency media [I-D.ietf-moq-transport]. MOQT defines a publish/subscribe media delivery layer across set of participating relays for supporting wide range of use-cases with different resiliency and latency (live, interactive) needs without compromising the scalability and cost effectiveness associated with content delivery networks. It supports sending media objects through sets of relays nodes.

Typically a MOQ Relay doesn’t need to access the media content, thus allowing the media to be "end-to-end" encrypted so that it cannot be decrypted by the relays. However for a relay to participate effectively in the media delivery, it needs to access naming information of a MOQT object to carryout the required store and forward functions.

As such, two layers of security are required:

1. Hop-by-hop (HBH) security between two MOQT relays

2. End-to-end (E2E) security from the Original Publisher of an MOQT object to End Subscribers

The HBH security is provided by TLS in the QUIC connection that MOQT runs over. MOQT support different E2EE protection as well as allowing for E2EE security.

This document defines a scheme for E2E authenticated encryption of MOQT objects. This scheme is based on the SFrame mechanism for authenticated encryption of media objects [I-D.ietf-sframe-enc].

However, a secondary goal of this design is to minimize the amount of additional data the encryptions requires for each object. This is particularly important for very low bit rate audio applications where the encryption overhead can increase overall bandwidth usage by a significant percentage. To minimize the overhead added by end-to-end encryption, certain fields that would be redundant between MOQT and SFrame are not transmitted.

2. Terminology

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.

This document re

E2EE: End to End Encryption

HBH: Hop By Hop

3. MOQT Object Model Recap

MOQT defines a publish/subscribe based media delivery protocol, where in endpoints, called original publishers, publish objects which are delivered via participating relays to receiving endpoints, called end subscribers.

Section 2 of [I-D.ietf-moq-transport] defines hierarchical object model for application data, comprised of objects, groups and tracks.

Objects defines the basic data element, an addressable unit whose payload is sequence of bytes. All objects belong to a group, indicating ordering and potential dependencies. A track contains has collection of groups and serves as the entity against which a subscribers issue subscription requests.

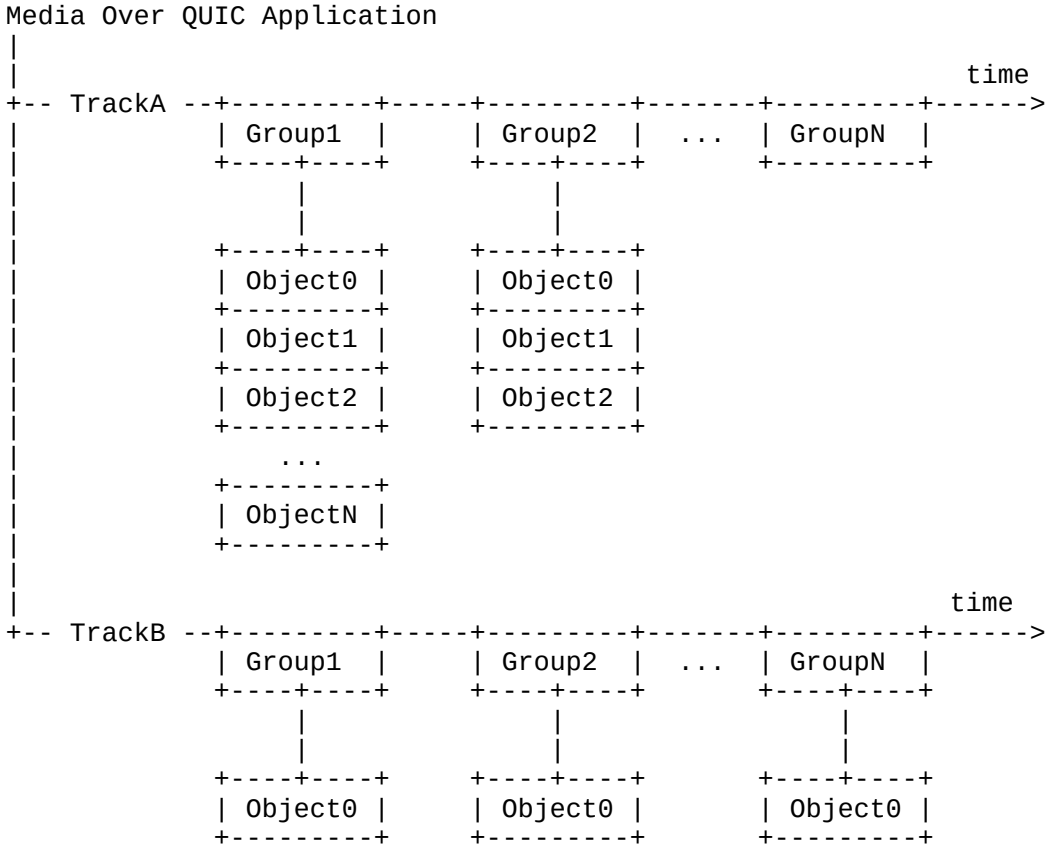


Figure 1: Structure of an MOQT session

Objects are comprised of two parts: envolope and a payload. The envelope is never end to end encrypted and is always visible to relays. The payload portion may be end to end encrypted, in which case it is only visible to the original publisher and the end subscribers. The application is solely responsible for the content of the object payload.

Tracks are identified by a combination of its TrackNamespace and TrackName. TrackNamespace and TrackName are treated as a sequence of binary bytes. Group and Objects are represented as variable length integers called GroupId and ObjectId respectively.

For purposes of this specification, we define FullTrackName as :

FullTrackName = TrackNamespace | TrackName

where | representations concatenation of byte strings,

and ObjectName is combination of following properties:

ObjectName = (FullTrackName, GroupId, ObjectId)

Two important properties of objects are:

1. ObjectNames are globally unique in a given relay network.

2. The data inside an object (and its size) can never change after the object is first published. There can never be two objects with the same name but different data.

One of the ways system keep the object names unique is by using a fully qualified domain names or UUIDs as part of the TrackNamespace.

4. Secure Objects

Section 8.1.1 of [I-D.ietf-moq-transport] defines fields of a canonical MOQT Object. The protection scheme defined in this draft encrypts the Object Payload and authenticates the Group ID, Object ID, and Object Payload fields, regardless of the on-the-wire encoding of the objects over QUIC Datagrams or QUIC streams.

4.1. Setup Assumptions

We assume that the application assigns each track a set of (Key ID, track_base_key) tuples, where each track_base_key is known only to authorized original publishers and end subscribers for a given track. How these per-track secrets are established is outside the scope of this specification. We also assume that the application defines which Key ID should be used for a given encryption operation (For decryption, the Key ID is obtained from the object payload).

It is also up to the application to specify the ciphersuite to be used for each track’s encryption context. Any SFrame ciphersuite can be used.

4.2. Secure Object Format

The payload of a secure object comprises an AEAD-encrypted object payload, with object header extension that specifies the Key ID in use.

```
SECURE_OBJECT {
  Key ID (i),
  Encrypted Data (..),
}
```

4.3. Encryption Schema

MOQT secure object protection relies on an SFrame cipher suite to define the AEAD encryption algorithm and hash algorithm in use [RFC9605]. We will refer to the following aspects of the AEAD and the hash algorithm below:

* AEAD.Encrypt and AEAD.Decrypt - The encryption and decryption functions for the AEAD. We follow the convention of RFC 5116 [RFC5116] and consider the authentication tag part of the ciphertext produced by AEAD.Encrypt (as opposed to a separate field as in SRTP [RFC3711]).

* AEAD.Nk - The size in bytes of a key for the encryption algorithm

* AEAD.Nn - The size in bytes of a nonce for the encryption algorithm

* AEAD.Nt - The overhead in bytes of the encryption algorithm (typically the size of a "tag" that is added to the plaintext)

* AEAD.Nka - For cipher suites using the compound AEAD described in Section 4.5.1 of [RFC9605], the size in bytes of a key for the underlying encryption algorithm

* Hash.Nh - The size in bytes of the output of the hash function

4.4. Metadata Authentication

The Key ID, FullTrackName, Group ID, and Object ID for a given object are authenticated as part of secure object encryption. This ensures, for example, that encrypted objects cannot be replayed across tracks.

When protecting or unprotecting a secure object, an endpoint encodes the Key ID, Group ID, Object ID, and FullTrackName in the following data structure, for input to the AEAD function’s AAD argument:

```
SECURE_OBJECT_AAD {
    Key ID (i),
    Group ID (i),
    Object ID (i),
    Track Namespace (tuple),
    Track Name (b),
}
```

4.5. Nonce Formation

The Group ID and Object ID for an object are used to form a 96-bit counter (CTR) value, which XORed with a salt to form the nonce used in AEAD encryption. The counter value is formed by encoding the Group ID and Object ID as QUIC varints, then concatenating these representations. This scheme MUST NOT be applied to an object where group ID is larger than 2^62 or the object ID is larger than 2^30.

```
def encode_varint(x):
    if x < 0x40:
        return (x, 8)
    elif x < 0x4000:
        return (0x4000 + x, 16)
    elif x < 0x40000000:
        return (0x80000000 + x, 32)
    elif x < 0x4000000000000000:
        return (0xc000000000000000 + x, 64)

def encode_ctr(group_id, object_id):
    (group_id, group_bits) = encode_varint(group_id)
    (object_id, object_bits) = encode_varint(object_id)

    group_shift = 96 - group_bits
    object_shift = group_shift - object_bits

    return (group_id << group_shift) | (object_id << object_shift)
```

4.6. Key Derivation

Encryption and decryption use a key and salt derived from the track_base_key associated with a Key ID. Given a track_base_key value, the key and salt are derived using HMAC-based Key Derivation Function (HKDF) [RFC5869] as follows:

```
def derive_key_salt(Key ID, track_base_key):
  moq_secret = HKDF-Extract("", track_base_key)
  moq_key_label = "MOQ 1.0 Secret key " + Key ID + cipher_suite
  moq_key =
    HKDF-Expand(moq_secret, moq_key_label, AEAD.Nk)

  moq_salt_label = "MOQ 1.0 Secret salt " + Key ID + cipher_suite
  moq_salt =
    HKDF-Expand(moq_secret, moq_salt_label, AEAD.Nn)

  return moq_key, moq_salt
```

In the derivation of moq_secret:

* The + operator represents concatenation of byte strings.

* The Key ID value is encoded as an 8-byte big-endian integer.

* The cipher_suite value is a 2-byte big-endian integer representing the cipher suite in use (see [I-D.ietf-sframe-enc]).

The hash function used for HKDF is determined by the cipher suite in use.

4.7. Encryption

MOQT secure object encryption uses the AEAD encryption algorithm for the cipher suite in use. The key for the encryption is the moq_key derived from the track_base_key Section 4.6. The nonce is formed by first XORing the moq_salt with the current CTR value, and then encoding the result as a big-endian integer of length AEAD.Nn.

The payload field from the MOQT object is used by the AEAD algorithm for the plaintext.

The encryptor forms an SecObj header using the Key ID value provided.

The encryption procedure is as follows:

1. From the MOQT Object obtain MOQT object payload as the plaintext to encrypt. Get the GroupId and ObjectId from the MOQT object envelope.

2. Retrieve the moq_key and moq_salt matching the Key ID.

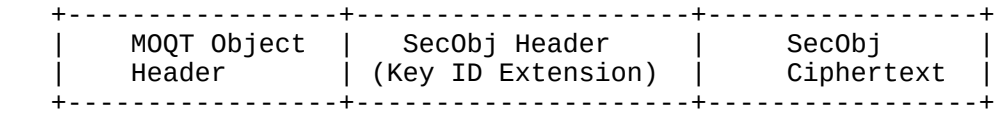
3. Form the aad input as described in Section 4.4.

4. Form the nonce by as described in Section 4.5.

5. Apply the AEAD encryption function with moq_key, nonce, aad and object payload as inputs.

6. Add the Key ID value to Key ID Object Header Extension.

The final SecureObject is formed from the MOQT transport headers, then the Key ID encdoded as QUIC variale length integer[RFC9000], followed by the output of the encryption.



Below shows psuedocode for the encryption process.

```
def encrypt(full_track_name, key_id, object):
    # Identify the appropriate encryption context
    ctx = context_for_track(full_track_name)
    moq_key, moq_salt = ctx.key_store[key_id]

    # Compute the required CTR parameter
    ctr = encode_ctr(object.group_id, object.object_id)

    # Assemble the AAD value
    aad = encode_aad(key_id, ctr, full_track_name)

    # Perform the AEAD encryption
    nonce = xor(moq_salt, ctr)
    encrypted_payload = AEAD.encrypt(moq_key, nonce, aad, object.payload)

    # Assemble the secure object payload
    (encoded_kid, _) = encode_varint(key_id)
    object.payload = encoded_kid + encrypted_payload
```

4.8. Decryption

For decrypting, the Key ID from the header extension in the secure object is used to find the right key and salt for the encrypted object, the nonce field is obtained from the GroupId and ObjectId fields of the MOQT object header.

The decryption procedure is as follows:

1. Parse the SecureObject to obtain Key ID, the ciphertext corresponding to MOQT object payload and the GroupID and ObjectId from the MOQT object envelope.

2. Retrieve the moq_key and moq_salt matching the Key ID.

3. Form the aad input as described in Section 4.4.

4. Form the nonce by as described in Section 4.5.

5. Apply the AEAD decryption function with moq_key, nonce, aad and ciphertext as inputs.

Below shows psuedocode for the decrpytion process

```
def decrypt(full_track_name, object):
    # Parse the secure object to obtain key ID and ciphertext
    (kid, kid_byte_len) = (object.headers[kid_extension])
    ciphertext = object.payload[:]

    # Identify the appropriate encryption context for the full track name
    # and the key ID
    ctx = context_for_track(full_track_name)
    moq_key, moq_salt = ctx.key_store[kid]

    # Compute the required CTR parameter
    ctr = encode_ctr(object.group_id, object.object_id)

    # Assemble the AAD value
    aad = encode_aad(kid, ctr, full_track_name)

    # Perform the AEAD decryption
    nonce = xor(moq_salt, ctr)
    object.payload = AEAD.decrypt(moq_key, nonce, aad, ciphertext)
```

If a ciphertext fails to decrypt because there is no key available for the Key ID value presented, the client MAY buffer the ciphertext and retry decryption once a key with that Key ID is received. If a ciphertext fails to decrypt for any other reason, the client MUST discard the ciphertext. Invalid ciphertexts SHOULD be discarded in a way that is indistinguishable (to an external observer) from having processed a valid ciphertext. In other words, the decryption operation should take the same amount of time regardless of whether decryption succeeds or fails.

5. Security Considerations

The cryptographic computations described in this document are exactly those performed in the SFrame encryption scheme defined in [I-D.ietf-sframe-enc], The scheme in this document is effectively a "virtualized" version of SFrame:

* The CTR value used in nonce formation is not carried in the object payload, but instead synthesized from the GroupID and ObjectID.

* The AAD for the AEAD operation is not sent on the wire (as with the SFrame Header), but constructed locally by the encrypting and decrypting endpoints.

* The format of the AAD is different:

  - The SFrame Header is constructed using QUIC-style varints, instead of the variable-length integer scheme defined in SFrame.

  - The GroupID and GroupID are sent directly, not as the packed CTR value.

* The metadata input in to SFrame operations is defined to be the FullTrackName value for the object.

* The labels used in key derivation reflect MOQ usage, not generic SFrame.

The security considerations discussed in the SFrame specification thus also apply here.

The SFrame specification lists several things that an application needs to account for in order to use SFrame securely, which are all accounted for here:

1. *Header value uniqueness:* Uniqueness of CTR values follows from the uniqueness of MOQT (GroupID, ObjectID) pairs. We only use one Key ID value, but instead use distinct SFrame contexts with distinct keys per track. This assures that the same (track_base_key, Key ID, CTR) tuple is never used twice.

2. *Key management:* We delegate this to the MOQT application, with subject to the assumptions described in Section 4.1.

3. *Anti-replay:* Replay is not possible within the MOQT framework because of the uniqueness constraints on ObjectIDs and objects, and because the group ID and object ID are cryptographically bound to the secure object payload.

4. *Metadata:* The analogue of the SFrame metadata input is defined in Section 4.4.

Any of the SFrame ciphersuites defined in the relevant IANA registry can be used to protect MOQT objects. The caution against short tags in Section 7.5 of [I-D.ietf-sframe-enc] still applies here, but the MOQT environment provides some safeguards that make it safer to use short tags, namely:

* MOQT has hop-by-hop protections provided by the underlying QUIC layer, so a brute-force attack could only be mounted by a relay.

* MOQT tracks have predictable object arrival rates, so a receiver can interpret a large deviation from this rate as a sign of an attack.

* The the binding of the secure object payload to other MOQT parameters (as metadata), together with MOQT's uniqueness properties ensure that a valid secure object payload cannot be replayed in a different context.

6. IANA Considerations

This document defines a new MOQT Object extension header for carrying Key ID value, under the MOQ Extension Headers registry.

Type	Value
0x0	Key ID Value - see Section 4.2

Table 1

7. References

7.1. Normative References

[I-D.ietf-moq-transport]
Curley, L., Pugin, K., Nandakumar, S., Vasiliev, V., and I. Swett, "Media over QUIC Transport", Work in Progress, Internet-Draft, draft-ietf-moq-transport-08, 12 February 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-moq-transport-08>.

[I-D.ietf-sframe-enc]
Omara, E., Uberti, J., Murillo, S. G., Barnes, R., and Y. Fablet, "Secure Frame (SFrame)", Work in Progress, Internet-Draft, draft-ietf-sframe-enc-09, 4 April 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-sframe-enc-09>.

[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>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/rfc/rfc5116>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/rfc/rfc5869>.

[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>.

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/rfc/rfc9000>.

[RFC9605] Omara, E., Uberti, J., Murillo, S. G., Barnes, R., Ed., and Y. Fablet, "Secure Frame (SFrame): Lightweight Authenticated Encryption for Real-Time Media", RFC 9605, DOI 10.17487/RFC9605, August 2024, <https://www.rfc-editor.org/rfc/rfc9605>.

7.2. Informative References

[RFC3711] Baugher, M., McGrew, D., Naslund, M., Carrara, E., and K. Norrman, "The Secure Real-time Transport Protocol (SRTP)", RFC 3711, DOI 10.17487/RFC3711, March 2004, <https://www.rfc-editor.org/rfc/rfc3711>.

Appendix A. Acknowledgements

TODO

Authors' Addresses

Cullen Jennings
Cisco
Email: fluffy@cisco.com

Suhas Nandakumar
Cisco
Email: snandaku@cisco.com

Richard Barnes
Cisco
Email: rlb@ipv.sx