QUIC
Internet-Draft
Intended status: Standards Track
Expires: November 23, 2018

M. Thomson, Ed.
Mozilla
S. Turner, Ed.
sn3rd
May 22, 2018

Using Transport Layer Security (TLS) to Secure QUIC
draft-ietf-quic-tls-12

Abstract

This document describes how Transport Layer Security (TLS) is used to secure QUIC.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic [1].

Working Group information can be found at https://github.com/quicwg [2]; source code and issues list for this draft can be found at https://github.com/quicwg/base-drafts/labels/-tls [3].

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/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on November 23, 2018.

Copyright Notice

Table of Contents

1. Introduction

This document describes how QUIC [QUIC-TRANSPORT] is secured using Transport Layer Security (TLS) version 1.3 [TLS13]. TLS 1.3 provides critical latency improvements for connection establishment over previous versions. Absent packet loss, most new connections can be established and secured within a single round trip; on subsequent connections between the same client and server, the client can often send application data immediately, that is, using a zero round trip setup.

This document describes how the standardized TLS 1.3 acts a security component of QUIC. The same design could work for TLS 1.2, though few of the benefits QUIC provides would be realized due to the handshake latency in versions of TLS prior to 1.3.

2. Notational Conventions

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 uses the terminology established in [QUIC-TRANSPORT].

For brevity, the acronym TLS is used to refer to TLS 1.3.

TLS terminology is used when referring to parts of TLS. Though TLS assumes a continuous stream of octets, it divides that stream into _records_. Most relevant to QUIC are the records that contain TLS _handshake messages_, which are discrete messages that are used for key agreement, authentication and parameter negotiation. Ordinarily, TLS records can also contain _application data_, though in the QUIC usage there is no use of TLS application data.

3. Protocol Overview

QUIC [QUIC-TRANSPORT] assumes responsibility for the confidentiality and integrity protection of packets. For this it uses keys derived from a TLS 1.3 connection [TLS13]; QUIC also relies on TLS 1.3 for authentication and negotiation of parameters that are critical to security and performance.

Rather than a strict layering, these two protocols are co-dependent: QUIC uses the TLS handshake; TLS uses the reliability and ordered delivery provided by QUIC streams.

This document defines how QUIC interacts with TLS. This includes a description of how TLS is used, how keying material is derived from TLS, and the application of that keying material to protect QUIC packets. Figure 1 shows the basic interactions between TLS and QUIC, with the QUIC packet protection being called out specially.

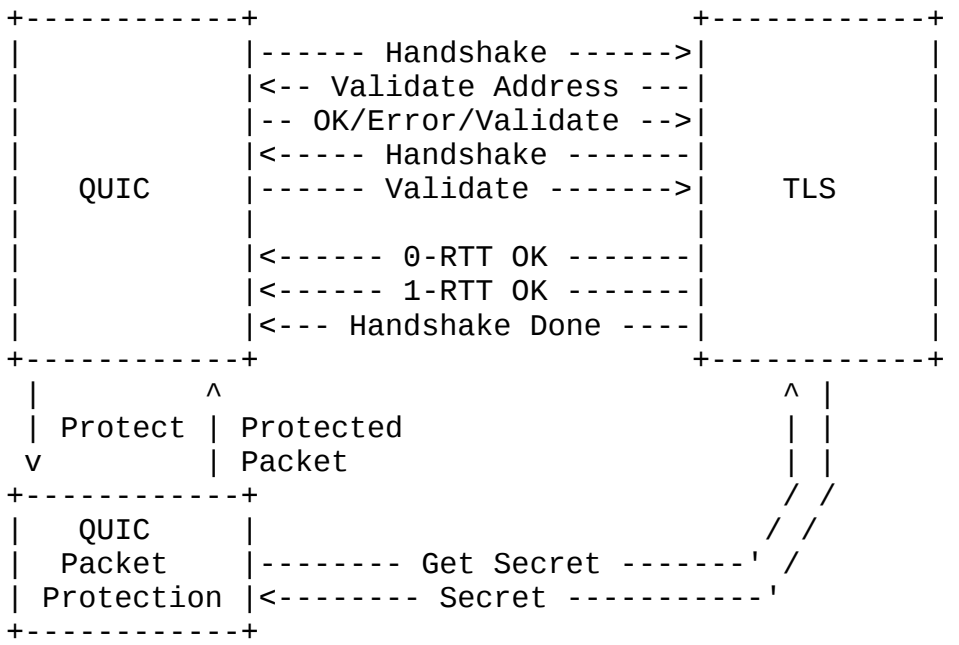

Figure 1: QUIC and TLS Interactions

The initial state of a QUIC connection has packets exchanged without any form of protection. In this state, QUIC is limited to using stream 0 and associated packets. Stream 0 is reserved for a TLS connection. This is a complete TLS connection as it would appear when layered over TCP; the only difference is that QUIC provides the reliability and ordering that would otherwise be provided by TCP.

At certain points during the TLS handshake, keying material is exported from the TLS connection for use by QUIC. This keying material is used to derive packet protection keys. Details on how and when keys are derived and used are included in Section 5.

3.1. TLS Overview

TLS provides two endpoints with a way to establish a means of communication over an untrusted medium (that is, the Internet) that ensures that messages they exchange cannot be observed, modified, or forged.

TLS features can be separated into two basic functions: an authenticated key exchange and record protection. QUIC primarily uses the authenticated key exchange provided by TLS but provides its own packet protection.

The TLS authenticated key exchange occurs between two entities: client and server. The client initiates the exchange and the server responds. If the key exchange completes successfully, both client and server will agree on a secret. TLS supports both pre-shared key (PSK) and Diffie-Hellman (DH) key exchanges. PSK is the basis for 0-RTT; the latter provides perfect forward secrecy (PFS) when the DH keys are destroyed.

After completing the TLS handshake, the client will have learned and authenticated an identity for the server and the server is optionally able to learn and authenticate an identity for the client. TLS supports X.509 [RFC5280] certificate-based authentication for both server and client.

The TLS key exchange is resistent to tampering by attackers and it produces shared secrets that cannot be controlled by either participating peer.

3.2. TLS Handshake

TLS 1.3 provides two basic handshake modes of interest to QUIC:

- A full 1-RTT handshake in which the client is able to send application data after one round trip and the server immediately responds after receiving the first handshake message from the client.

- A 0-RTT handshake in which the client uses information it has previously learned about the server to send application data immediately. This application data can be replayed by an attacker so it MUST NOT carry a self-contained trigger for any non-idempotent action.

A simplified TLS 1.3 handshake with 0-RTT application data is shown in Figure 2, see [TLS13] for more options and details.

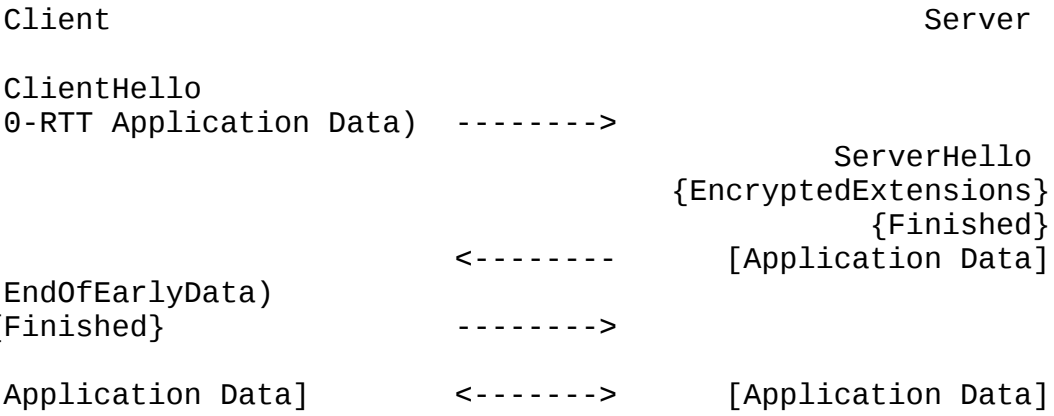

Figure 2: TLS Handshake with 0-RTT

This 0-RTT handshake is only possible if the client and server have previously communicated. In the 1-RTT handshake, the client is unable to send protected application data until it has received all of the handshake messages sent by the server.

Two additional variations on this basic handshake exchange are relevant to this document:

- The server can respond to a ClientHello with a HelloRetryRequest, which adds an additional round trip prior to the basic exchange. This is needed if the server wishes to request a different key exchange key from the client. HelloRetryRequest is also used to verify that the client is correctly able to receive packets on the address it claims to have (see [QUIC-TRANSPORT]).

- A pre-shared key mode can be used for subsequent handshakes to reduce the number of public key operations. This is the basis for 0-RTT data, even if the remainder of the connection is protected by a new Diffie-Hellman exchange.

4. TLS Usage

QUIC reserves stream 0 for a TLS connection. Stream 0 contains a complete TLS connection, which includes the TLS record layer. Other than the definition of a QUIC-specific extension (see Section 9.2), TLS is unmodified for this use. This means that TLS will apply confidentiality and integrity protection to its records. In particular, TLS record protection is what provides confidentiality protection for the TLS handshake messages sent by the server.

QUIC permits a client to send frames on streams starting from the first packet. The initial packet from a client contains a stream frame for stream 0 that contains the first TLS handshake messages

from the client. This allows the TLS handshake to start with the first packet that a client sends.

QUIC packets are protected using a scheme that is specific to QUIC, see Section 5. Keys are exported from the TLS connection when they become available using a TLS exporter (see Section 7.5 of [TLS13] and Section 5.3). After keys are exported from TLS, QUIC manages its own key schedule.

4.1. Handshake and Setup Sequence

The integration of QUIC with a TLS handshake is shown in more detail in Figure 3. QUIC "STREAM" frames on stream 0 carry the TLS handshake. QUIC performs loss recovery [QUIC-RECOVERY] for this stream and ensures that TLS handshake messages are delivered in the correct order.

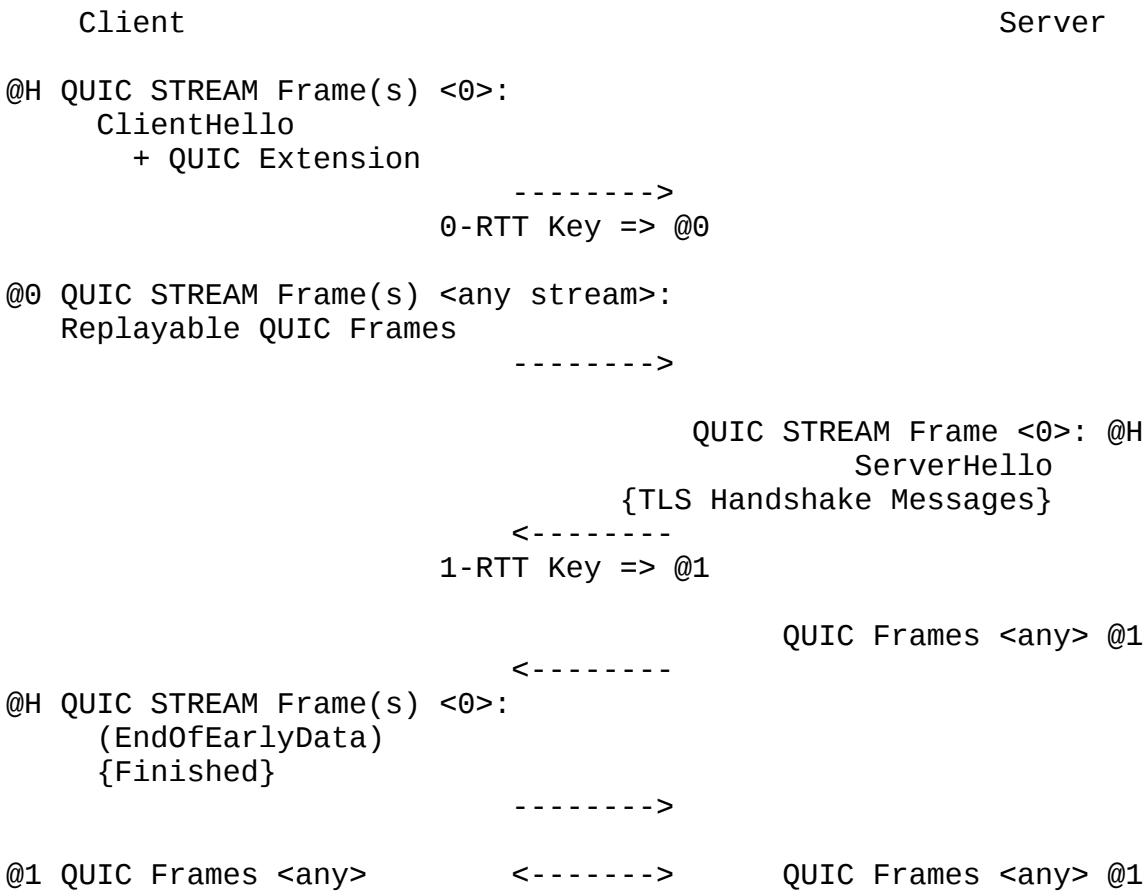

Figure 3: QUIC over TLS Handshake

In Figure 3, symbols mean:

- "<" and ">" enclose stream numbers.

- "@" indicates the keys that are used for protecting the QUIC packet (H = handshake, using keys from the well-known cleartext packet secret; 0 = 0-RTT keys; 1 = 1-RTT keys).

- "(" and ")" enclose messages that are protected with TLS 0-RTT handshake or application keys.

- "{" and "}" enclose messages that are protected by the TLS Handshake keys.

If 0-RTT is not attempted, then the client does not send packets protected by the 0-RTT key (@0). In that case, the only key transition on the client is from handshake packets (@H) to 1-RTT protection (@1), which happens after it sends its final set of TLS handshake messages.

Note: two different types of packet are used during the handshake by both client and server. The Initial packet carries a TLS ClientHello message; the remainder of the TLS handshake is carried in Handshake packets. The Retry packet carries a TLS HelloRetryRequest, if it is needed, and Handshake packets carry the remainder of the server handshake.

The server sends TLS handshake messages without protection (@H). The server transitions from no protection (@H) to full 1-RTT protection (@1) after it sends the last of its handshake messages.

Some TLS handshake messages are protected by the TLS handshake record protection. These keys are not exported from the TLS connection for use in QUIC. QUIC packets from the server are sent in the clear until the final transition to 1-RTT keys.

The client transitions from handshake (@H) to 0-RTT keys (@0) when sending 0-RTT data, and subsequently to to 1-RTT keys (@1) after its second flight of TLS handshake messages. This creates the potential for unprotected packets to be received by a server in close proximity to packets that are protected with 1-RTT keys.

More information on key transitions is included in Section 6.1.

4.2. Interface to TLS

As shown in Figure 1, the interface from QUIC to TLS consists of four primary functions: Handshake, Source Address Validation, Key Ready Events, and Secret Export.

Additional functions might be needed to configure TLS.

4.2.1. Handshake Interface

In order to drive the handshake, TLS depends on being able to send and receive handshake messages on stream 0. There are two basic functions on this interface: one where QUIC requests handshake messages and one where QUIC provides handshake packets.

Before starting the handshake QUIC provides TLS with the transport parameters (see Section 9.2) that it wishes to carry.

A QUIC client starts TLS by requesting TLS handshake octets from TLS. The client acquires handshake octets before sending its first packet.

A QUIC server starts the process by providing TLS with stream 0 octets.

Each time that an endpoint receives data on stream 0, it delivers the octets to TLS if it is able. Each time that TLS is provided with new data, new handshake octets are requested from TLS. TLS might not provide any octets if the handshake messages it has received are incomplete or it has no data to send.

At the server, when TLS provides handshake octets, it also needs to indicate whether the octets contain a HelloRetryRequest. A HelloRetryRequest MUST always be sent in a Retry packet, so the QUIC server needs to know whether the octets are a HelloRetryRequest.

Once the TLS handshake is complete, this is indicated to QUIC along with any final handshake octets that TLS needs to send. TLS also provides QUIC with the transport parameters that the peer advertised during the handshake.

Once the handshake is complete, TLS becomes passive. TLS can still receive data from its peer and respond in kind, but it will not need to send more data unless specifically requested - either by an application or QUIC. One reason to send data is that the server might wish to provide additional or updated session tickets to a client.

When the handshake is complete, QUIC only needs to provide TLS with any data that arrives on stream 0. In the same way that is done during the handshake, new data is requested from TLS after providing received data.

Important: Until the handshake is reported as complete, the connection and key exchange are not properly authenticated at the

server. Even though 1-RTT keys are available to a server after receiving the first handshake messages from a client, the server cannot consider the client to be authenticated until it receives and validates the client's Finished message.

The requirement for the server to wait for the client Finished message creates a dependency on that message being delivered. A client can avoid the potential for head-of-line blocking that this implies by sending a copy of the STREAM frame that carries the Finished message in multiple packets. This enables immediate server processing for those packets.

4.2.2. Source Address Validation

During the processing of the TLS ClientHello, TLS requests that the transport make a decision about whether to request source address validation from the client.

An initial TLS ClientHello that resumes a session includes an address validation token in the session ticket; this includes all attempts at 0-RTT. If the client does not attempt session resumption, no token will be present. While processing the initial ClientHello, TLS provides QUIC with any token that is present. In response, QUIC provides one of three responses:

- proceed with the connection,
- ask for client address validation, or
- abort the connection.

If QUIC requests source address validation, it also provides a new address validation token. TLS includes that along with any information it requires in the cookie extension of a TLS HelloRetryRequest message. In the other cases, the connection either proceeds or terminates with a handshake error.

The client echoes the cookie extension in a second ClientHello. A ClientHello that contains a valid cookie extension will always be in response to a HelloRetryRequest. If address validation was requested by QUIC, then this will include an address validation token. TLS makes a second address validation request of QUIC, including the value extracted from the cookie extension. In response to this request, QUIC cannot ask for client address validation, it can only abort or permit the connection attempt to proceed.

QUIC can provide a new address validation token for use in session resumption at any time after the handshake is complete. Each time a

new token is provided TLS generates a NewSessionTicket message, with the token included in the ticket.

See Section 7 for more details on client address validation.

4.2.3. Key Ready Events

TLS provides QUIC with signals when 0-RTT and 1-RTT keys are ready for use. These events are not asynchronous, they always occur immediately after TLS is provided with new handshake octets, or after TLS produces handshake octets.

When TLS completed its handshake, 1-RTT keys can be provided to QUIC. On both client and server, this occurs after sending the TLS Finished message.

This ordering means that there could be frames that carry TLS handshake messages ready to send at the same time that application data is available. An implementation MUST ensure that TLS handshake messages are always sent in packets protected with handshake keys (see Section 5.3.2). Separate packets are required for data that needs protection from 1-RTT keys.

If 0-RTT is possible, it is ready after the client sends a TLS ClientHello message or the server receives that message. After providing a QUIC client with the first handshake octets, the TLS stack might signal that 0-RTT keys are ready. On the server, after receiving handshake octets that contain a ClientHello message, a TLS server might signal that 0-RTT keys are available.

1-RTT keys are used for packets in both directions. 0-RTT keys are only used to protect packets sent by the client.

4.2.4. Secret Export

Details how secrets are exported from TLS are included in Section 5.3.

4.2.5. TLS Interface Summary

Figure 4 summarizes the exchange between QUIC and TLS for both client and server.

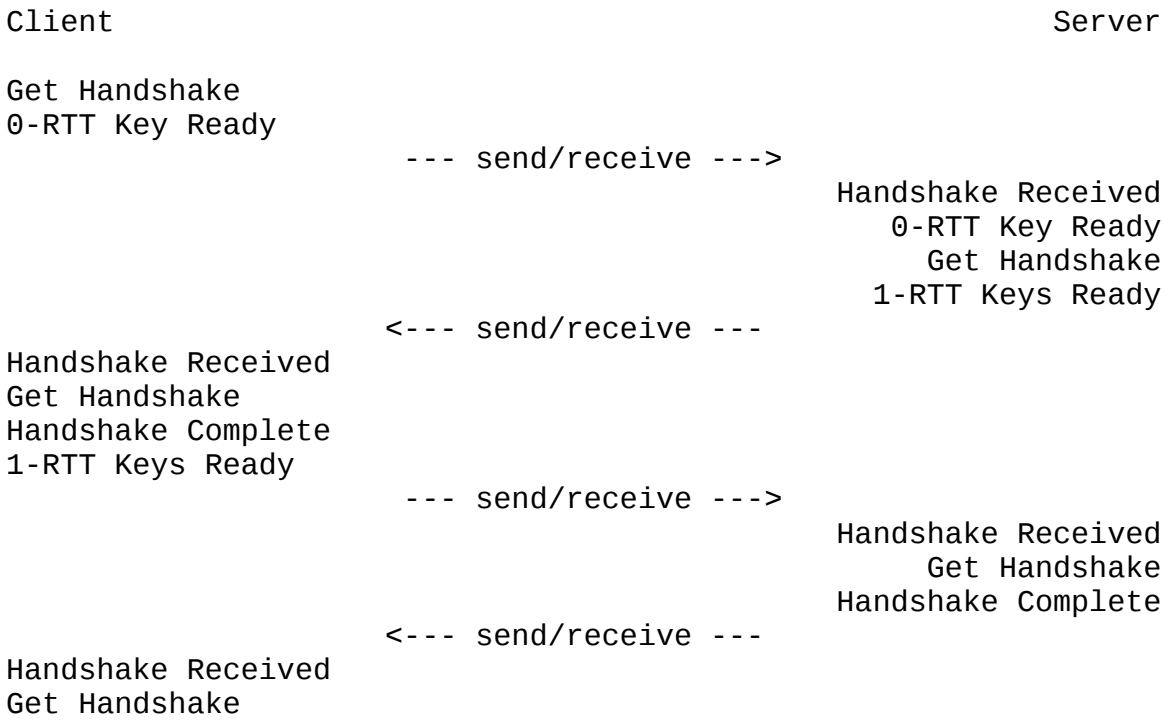

Figure 4: Interaction Summary between QUIC and TLS

4.3. TLS Version

This document describes how TLS 1.3 [TLS13] is used with QUIC.

In practice, the TLS handshake will negotiate a version of TLS to use. This could result in a newer version of TLS than 1.3 being negotiated if both endpoints support that version. This is acceptable provided that the features of TLS 1.3 that are used by QUIC are supported by the newer version.

A badly configured TLS implementation could negotiate TLS 1.2 or another older version of TLS. An endpoint MUST terminate the connection if a version of TLS older than 1.3 is negotiated.

4.4. ClientHello Size

QUIC requires that the initial handshake packet from a client fit within the payload of a single packet. The size limits on QUIC packets mean that a record containing a ClientHello needs to fit within 1129 octets, though endpoints can reduce the size of their connection ID to increase by up to 22 octets.

A TLS ClientHello can fit within this limit with ample space remaining. However, there are several variables that could cause

this limit to be exceeded. Implementations are reminded that large session tickets or HelloRetryRequest cookies, multiple or large key shares, and long lists of supported ciphers, signature algorithms, versions, QUIC transport parameters, and other negotiable parameters and extensions could cause this message to grow.

For servers, the size of the session tickets and HelloRetryRequest cookie extension can have an effect on a client's ability to connect. Choosing a small value increases the probability that these values can be successfully used by a client.

The TLS implementation does not need to ensure that the ClientHello is sufficiently large. QUIC PADDING frames are added to increase the size of the packet as necessary.

4.5. Peer Authentication

The requirements for authentication depend on the application protocol that is in use. TLS provides server authentication and permits the server to request client authentication.

A client MUST authenticate the identity of the server. This typically involves verification that the identity of the server is included in a certificate and that the certificate is issued by a trusted entity (see for example [RFC2818]).

A server MAY request that the client authenticate during the handshake. A server MAY refuse a connection if the client is unable to authenticate when requested. The requirements for client authentication vary based on application protocol and deployment.

A server MUST NOT use post-handshake client authentication (see Section 4.6.2 of [TLS13]).

4.6. Rejecting 0-RTT

A server rejects 0-RTT by rejecting 0-RTT at the TLS layer. This results in early exporter keys being unavailable, thereby preventing the use of 0-RTT for QUIC.

A client that attempts 0-RTT MUST also consider 0-RTT to be rejected if it receives a Retry or Version Negotiation packet.

When 0-RTT is rejected, all connection characteristics that the client assumed might be incorrect. This includes the choice of application protocol, transport parameters, and any application configuration. The client therefore MUST reset the state of all streams, including application state bound to those streams.

4.7. TLS Errors

Errors in the TLS connection SHOULD be signaled using TLS alerts on stream 0. A failure in the handshake MUST be treated as a QUIC connection error of type TLS_HANDSHAKE_FAILED. Once the handshake is complete, an error in the TLS connection that causes a TLS alert to be sent or received MUST be treated as a QUIC connection error of type TLS_FATAL_ALERT_GENERATED or TLS_FATAL_ALERT_RECEIVED respectively.

5. QUIC Packet Protection

QUIC packet protection provides authenticated encryption of packets. This provides confidentiality and integrity protection for the content of packets (see Section 5.4). Packet protection uses keys that are exported from the TLS connection (see Section 5.3).

Different keys are used for QUIC packet protection and TLS record protection. TLS handshake messages are protected solely with TLS record protection, but post-handshake messages are redundantly protected with both the QUIC packet protection and the TLS record protection. These messages are limited in number, and so the additional overhead is small.

5.1. Installing New Keys

As TLS reports the availability of keying material, the packet protection keys and initialization vectors (IVs) are updated (see Section 5.3). The selection of AEAD function is also updated to match the AEAD negotiated by TLS.

For packets other than any handshake packets (see Section 6.1), once a change of keys has been made, packets with higher packet numbers MUST be sent with the new keying material. The KEY_PHASE bit on these packets is inverted each time new keys are installed to signal the use of the new keys to the recipient (see Section 6 for details).

An endpoint retransmits stream data in a new packet. New packets have new packet numbers and use the latest packet protection keys. This simplifies key management when there are key updates (see Section 6.2).

5.2. Enabling 0-RTT

In order to be usable for 0-RTT, TLS MUST provide a NewSessionTicket message that contains the "max_early_data" extension with the value 0xffffffff; the amount of data which the client can send in 0-RTT is controlled by the "initial_max_data" transport parameter supplied by

the server. A client MUST treat receipt of a NewSessionTicket that contains a "max_early_data" extension with any other value as a connection error of type PROTOCOL_VIOLATION.

Early data within the TLS connection MUST NOT be used. As it is for other TLS application data, a server MUST treat receiving early data on the TLS connection as a connection error of type PROTOCOL_VIOLATION.

5.3. QUIC Key Expansion

QUIC uses a system of packet protection secrets, keys and IVs that are modelled on the system used in TLS [TLS13]. The secrets that QUIC uses as the basis of its key schedule are obtained using TLS exporters (see Section 7.5 of [TLS13]).

5.3.1. QHKDF-Expand

QUIC uses the Hash-based Key Derivation Function (HKDF) [HKDF] with the same hash function negotiated by TLS for key derivation. For example, if TLS is using the TLS_AES_128_GCM_SHA256, the SHA-256 hash function is used.

Most key derivations in this document use the QHKDF-Expand function, which uses the HKDF expand function and is modelled on the HKDF-Expand-Label function from TLS 1.3 (see Section 7.1 of [TLS13]). QHKDF-Expand differs from HKDF-Expand-Label in that it uses a different base label and omits the Context argument.

```
QHKDF-Expand(Secret, Label, Length) =
   HKDF-Expand(Secret, QhkdfExpandInfo, Length)
```

The HKDF-Expand function used by QHKDF-Expand uses the PRF hash function negotiated by TLS, except for handshake secrets and keys derived from them (see Section 5.3.2).

Where the "info" parameter of HKDF-Expand is an encoded "QhkdfExpandInfo" structure:

```
struct {
   uint16 length = Length;
   opaque label<6..255> = "QUIC " + Label;
} QhkdfExpandInfo;
```

For example, assuming a hash function with a 32 octet output, derivation for a client packet protection key would use HKDF-Expand with an "info" parameter of 0x00200851554943206b6579.

5.3.2. Handshake Secrets

Packets that carry the TLS handshake (Initial, Retry, and Handshake) are protected with a secret derived from the Destination Connection ID field from the client's Initial packet. Specifically:

```
handshake_salt = 0x9c108f98520a5c5c32968e950e8a2c5fe06d6c38
handshake_secret =
    HKDF-Extract(handshake_salt, client_dst_connection_id)

client_handshake_secret =
   QHKDF-Expand(handshake_secret, "client hs", Hash.length)
server_handshake_secret =
   QHKDF-Expand(handshake_secret, "server hs", Hash.length)
```

The hash function for HKDF when deriving handshake secrets and keys is SHA-256 [SHA]. The connection ID used with QHKDF-Expand is the connection ID chosen by the client.

The handshake salt is a 20 octet sequence shown in the figure in hexadecimal notation. Future versions of QUIC SHOULD generate a new salt value, thus ensuring that the keys are different for each version of QUIC. This prevents a middlebox that only recognizes one version of QUIC from seeing or modifying the contents of handshake packets from future versions.

Note: The Destination Connection ID is of arbitrary length, and it could be zero length if the server sends a Retry packet with a zero-length Source Connection ID field. In this case, the handshake keys provide no assurance to the client that the server received its packet; the client has to rely on the exchange that included the Retry packet for that property.

5.3.3. 0-RTT Secret

0-RTT keys are those keys that are used in resumed connections prior to the completion of the TLS handshake. Data sent using 0-RTT keys might be replayed and so has some restrictions on its use, see Section 8.2. 0-RTT keys are used after sending or receiving a ClientHello.

The secret is exported from TLS using the exporter label "EXPORTER-QUIC 0rtt" and an empty context. The size of the secret MUST be the size of the hash output for the PRF hash function negotiated by TLS. This uses the TLS early_exporter_secret. The QUIC 0-RTT secret is only used for protection of packets sent by the client.

```
client_0rtt_secret =
   TLS-Early-Exporter("EXPORTER-QUIC 0rtt", "", Hash.length)
```

5.3.4. 1-RTT Secrets

1-RTT keys are used by both client and server after the TLS handshake completes. There are two secrets used at any time: one is used to derive packet protection keys for packets sent by the client, the other for packet protection keys on packets sent by the server.

The initial client packet protection secret is exported from TLS using the exporter label "EXPORTER-QUIC client 1rtt"; the initial server packet protection secret uses the exporter label "EXPORTER-QUIC server 1rtt". Both exporters use an empty context. The size of the secret MUST be the size of the hash output for the PRF hash function negotiated by TLS.

```
client_pp_secret<0> =
   TLS-Exporter("EXPORTER-QUIC client 1rtt", "", Hash.length)
server_pp_secret<0> =
   TLS-Exporter("EXPORTER-QUIC server 1rtt", "", Hash.length)
```

These secrets are used to derive the initial client and server packet protection keys.

5.3.5. Updating 1-RTT Secrets

After a key update (see Section 6.2), the 1-RTT secrets are updated using QHKDF-Expand. Updated secrets are derived from the existing packet protection secret. A Label parameter of "client 1rtt" is used for the client secret and "server 1rtt" for the server. The Length is the same as the native output of the PRF hash function.

```
client_pp_secret<N+1> =
  QHKDF-Expand(client_pp_secret<N>, "client 1rtt", Hash.length)
server_pp_secret<N+1> =
  QHKDF-Expand(server_pp_secret<N>, "server 1rtt", Hash.length)
```

This allows for a succession of new secrets to be created as needed.

5.3.6. Packet Protection Keys

The complete key expansion uses a similar process for key expansion to that defined in Section 7.3 of [TLS13], using QHKDF-Expand in place of HKDF-Expand-Label. QUIC uses the AEAD function negotiated by TLS.

The packet protection key and IV used to protect the 0-RTT packets sent by a client are derived from the QUIC 0-RTT secret. The packet protection keys and IVs for 1-RTT packets sent by the client and server are derived from the current generation of client and server 1-RTT secrets (client_pp_secret<i> and server_pp_secret<i>) respectively.

The length of the QHKDF-Expand output is determined by the requirements of the AEAD function selected by TLS. The key length is the AEAD key size. As defined in Section 5.3 of [TLS13], the IV length is the larger of 8 or N_MIN (see Section 4 of [AEAD]; all ciphersuites defined in [TLS13] have N_MIN set to 12).

The size of the packet protection key is determined by the packet protection algorithm, see Section 5.6.

For any secret S, the AEAD key uses a label of "key", the IV uses a label of "iv", packet number encryption uses a label of "pn":

```
key = QHKDF-Expand(S, "key", key_length)
iv = QHKDF-Expand(S, "iv", iv_length)
pn_key = QHKDF-Expand(S, "pn", pn_key_length)
```

Separate keys are derived for packet protection by clients and servers. Each endpoint uses the packet protection key of its peer to remove packet protection. For example, client packet protection keys and IVs - which are also used by the server to remove the protection added by a client - for AEAD_AES_128_GCM are derived from 1-RTT secrets as follows:

```
client_pp_key<i> = QHKDF-Expand(client_pp_secret<i>, "key", 16)
client_pp_iv<i>  = QHKDF-Expand(client_pp_secret<i>, "iv", 12)
client_pp_pn<i>  = QHKDF-Expand(client_pp_secret<i>, "pn", 12)
```

The QUIC packet protection initially starts with keying material derived from handshake keys. For a client, when the TLS state machine reports that the ClientHello has been sent, 0-RTT keys can be generated and installed for writing, if 0-RTT is available. Finally, the TLS state machine reports completion of the handshake and 1-RTT keys can be generated and installed for writing.

5.4. QUIC AEAD Usage

The Authentication Encryption with Associated Data (AEAD) [AEAD] function used for QUIC packet protection is AEAD that is negotiated for use with the TLS connection. For example, if TLS is using the TLS_AES_128_GCM_SHA256, the AEAD_AES_128_GCM function is used.

QUIC packets are protected prior to applying packet number encryption (Section 5.6). The unprotected packet number is part of the associated data (A). When removing packet protection, an endpoint first removes the protection from the packet number.

All QUIC packets other than Version Negotiation and Stateless Reset packets are protected with an AEAD algorithm [AEAD]. Prior to establishing a shared secret, packets are protected with AEAD_AES_128_GCM and a key derived from the client's connection ID (see Section 5.3.2). This provides protection against off-path attackers and robustness against QUIC version unaware middleboxes, but not against on-path attackers.

All ciphersuites currently defined for TLS 1.3 - and therefore QUIC - have a 16-byte authentication tag and produce an output 16 bytes larger than their input.

Once TLS has provided a key, the contents of regular QUIC packets immediately after any TLS messages have been sent are protected by the AEAD selected by TLS.

The key, K, is either the client packet protection key (client_pp_key<i>) or the server packet protection key (server_pp_key<i>), derived as defined in Section 5.3.

The nonce, N, is formed by combining the packet protection IV (either client_pp_iv<i> or server_pp_iv<i>) with the packet number. The 64 bits of the reconstructed QUIC packet number in network byte order is left-padded with zeros to the size of the IV. The exclusive OR of the padded packet number and the IV forms the AEAD nonce.

The associated data, A, for the AEAD is the contents of the QUIC header, starting from the flags octet in either the short or long header.

The input plaintext, P, for the AEAD is the content of the QUIC frame following the header, as described in [QUIC-TRANSPORT].

The output ciphertext, C, of the AEAD is transmitted in place of P.

5.5. Packet Numbers

QUIC has a single, contiguous packet number space. In comparison, TLS restarts its sequence number each time that record protection keys are changed. The sequence number restart in TLS ensures that a compromise of the current traffic keys does not allow an attacker to truncate the data that is sent after a key update by sending

additional packets under the old key (causing new packets to be discarded).

QUIC does not assume a reliable transport and is required to handle attacks where packets are dropped in other ways. QUIC is therefore not affected by this form of truncation.

The QUIC packet number is not reset and it is not permitted to go higher than its maximum value of 2^62-1. This establishes a hard limit on the number of packets that can be sent.

Some AEAD functions have limits for how many packets can be encrypted under the same key and IV (see for example [AEBounds]). This might be lower than the packet number limit. An endpoint MUST initiate a key update (Section 6.2) prior to exceeding any limit set for the AEAD that is in use.

TLS maintains a separate sequence number that is used for record protection on the connection that is hosted on stream 0. This sequence number is not visible to QUIC.

5.6. Packet Number Protection

QUIC packets are protected using a key that is derived from the current set of secrets. The key derived using the "pn" label is used to protect the packet number from casual observation. The packet number protection algorithm depends on the negotiated AEAD.

Packet number protection is applied after packet protection is applied (see Section 5.4). The ciphertext of the packet is sampled and used as input to an encryption algorithm.

In sampling the packet ciphertext, the packet number length is assumed to be the smaller of the maximum possible packet number encoding (4 octets), or the size of the protected packet minus the minimum expansion for the AEAD. For example, the sampled ciphertext for a packet with a short header can be determined by:

"sample_offset = min(1 + connection_id_length + 4, packet_length - aead_expansion) sample = packet[sample_offset..sample_offset+sample_length] "

To ensure that this process does not sample the packet number, packet number protection algorithms MUST NOT sample more ciphertext than the minimum expansion of the corresponding AEAD.

Packet number protection is applied to the packet number encoded as described in Section 4.8 of [QUIC-TRANSPORT]. Since the length of

the packet number is stored in the first octet of the encoded packet number, it may be necessary to progressively decrypt the packet number.

Before a TLS ciphersuite can be used with QUIC, a packet protection algorithm MUST be specifed for the AEAD used with that ciphersuite. This document defines algorithms for AEAD_AES_128_GCM, AEAD_AES_128_CCM, AEAD_AES_256_GCM, AEAD_AES_256_CCM (all AES AEADs are defined in [RFC5116]), and AEAD_CHACHA20_POLY1305 ([CHACHA]).

5.6.1. AES-Based Packet Number Protection

This section defines the packet protection algorithm for AEAD_AES_128_GCM, AEAD_AES_128_CCM, AEAD_AES_256_GCM, and AEAD_AES_256_CCM. AEAD_AES_128_GCM and AEAD_AES_128_CCM use 128-bit AES [AES] in counter (CTR) mode. AEAD_AES_256_GCM, and AEAD_AES_256_CCM use 256-bit AES in CTR mode.

This algorithm samples 16 octets from the packet ciphertext. This value is used as the counter input to AES-CTR.

```
encrypted_pn = AES-CTR(pn_key, sample, packet_number)
```

5.6.2. ChaCha20-Based Packet Number Protection

When AEAD_CHACHA20_POLY1305 is in use, packet number protection uses the raw ChaCha20 function as defined in Section 2.4 of [CHACHA]. This uses a 256-bit key and 16 octets sampled from the packet protection output.

The first 4 octets of the sampled ciphertext are interpreted as a 32-bit number in little-endian order and are used as the block count. The remaining 12 octets are interpreted as three concatenated 32-bit numbers in little-endian order and used as the nonce.

The encoded packet number is then encrypted with ChaCha20 directly. In pseudocode:

```
counter = DecodeLE(sample[0..3])
nonce = DecodeLE(sample[4..7], sample[8..11], sample[12..15])
encrypted_pn = ChaCha20(pn_key, counter, nonce, packet_number)
```

5.7. Receiving Protected Packets

Once an endpoint successfully receives a packet with a given packet number, it MUST discard all packets with higher packet numbers if they cannot be successfully unprotected with either the same key, or - if there is a key update - the next packet protection key (see

Section 6.2). Similarly, a packet that appears to trigger a key update, but cannot be unprotected successfully MUST be discarded.

Failure to unprotect a packet does not necessarily indicate the existence of a protocol error in a peer or an attack. The truncated packet number encoding used in QUIC can cause packet numbers to be decoded incorrectly if they are delayed significantly.

6. Key Phases

As TLS reports the availability of 0-RTT and 1-RTT keys, new keying material can be exported from TLS and used for QUIC packet protection. At each transition during the handshake a new secret is exported from TLS and packet protection keys are derived from that secret.

Every time that a new set of keys is used for protecting outbound packets, the KEY_PHASE bit in the public flags is toggled. 0-RTT protected packets use the QUIC long header, they do not use the KEY_PHASE bit to select the correct keys (see Section 6.1.1).

Once the connection is fully enabled, the KEY_PHASE bit allows a recipient to detect a change in keying material without necessarily needing to receive the first packet that triggered the change. An endpoint that notices a changed KEY_PHASE bit can update keys and decrypt the packet that contains the changed bit, see Section 6.2.

The KEY_PHASE bit is included as the 0x20 bit of the QUIC short header.

Transitions between keys during the handshake are complicated by the need to ensure that TLS handshake messages are sent with the correct packet protection.

6.1. Packet Protection for the TLS Handshake

The initial exchange of packets that carry the TLS handshake are AEAD-protected using the handshake secrets generated as described in Section 5.3.2. All TLS handshake messages up to the TLS Finished message sent by either endpoint use packets protected with handshake keys.

Any TLS handshake messages that are sent after completing the TLS handshake do not need special packet protection rules. Packets containing these messages use the packet protection keys that are current at the time of sending (or retransmission).

Like the client, a server MUST send retransmissions of its unprotected handshake messages or acknowledgments for unprotected handshake messages sent by the client in packets protected with handshake keys.

6.1.1. Initial Key Transitions

Once the TLS handshake is complete, keying material is exported from TLS and used to protect QUIC packets.

Packets protected with 1-RTT keys initially have a KEY_PHASE bit set to 0. This bit inverts with each subsequent key update (see Section 6.2).

If the client sends 0-RTT data, it uses the 0-RTT packet type. The packet that contains the TLS EndOfEarlyData and Finished messages are sent in packets protected with handshake keys.

Using distinct packet types during the handshake for handshake messages, 0-RTT data, and 1-RTT data ensures that the server is able to distinguish between the different keys used to remove packet protection. All of these packets can arrive concurrently at a server.

A server might choose to retain 0-RTT packets that arrive before a TLS ClientHello. The server can then use those packets once the ClientHello arrives. However, the potential for denial of service from buffering 0-RTT packets is significant. These packets cannot be authenticated and so might be employed by an attacker to exhaust server resources. Limiting the number of packets that are saved might be necessary.

The server transitions to using 1-RTT keys after sending its first flight of TLS handshake messages, ending in the Finished. From this point, the server protects all packets with 1-RTT keys. Future packets are therefore protected with 1-RTT keys. Initially, these are marked with a KEY_PHASE of 0.

6.1.2. Retransmission and Acknowledgment of Unprotected Packets

TLS handshake messages from both client and server are critical to the key exchange. The contents of these messages determine the keys used to protect later messages. If these handshake messages are included in packets that are protected with these keys, they will be indecipherable to the recipient.

Even though newer keys could be available when retransmitting, retransmissions of these handshake messages MUST be sent in packets

protected with handshake keys. An endpoint MUST generate ACK frames for these messages and send them in packets protected with handshake keys.

A HelloRetryRequest handshake message might be used to reject an initial ClientHello. A HelloRetryRequest handshake message is sent in a Retry packet; any second ClientHello that is sent in response uses a Initial packet type. These packets are only protected with a predictable key (see Section 5.3.2). This is natural, because no shared secret will be available when these messages need to be sent. Upon receipt of a HelloRetryRequest, a client SHOULD cease any transmission of 0-RTT data; 0-RTT data will only be discarded by any server that sends a HelloRetryRequest.

The packet type ensures that protected packets are clearly distinguished from unprotected packets. Loss or reordering might cause unprotected packets to arrive once 1-RTT keys are in use, unprotected packets are easily distinguished from 1-RTT packets using the packet type.

Once 1-RTT keys are available to an endpoint, it no longer needs the TLS handshake messages that are carried in unprotected packets. However, a server might need to retransmit its TLS handshake messages in response to receiving an unprotected packet that contains ACK frames. A server MUST process ACK frames in unprotected packets until the TLS handshake is reported as complete, or it receives an ACK frame in a protected packet that acknowledges all of its handshake messages.

To limit the number of key phases that could be active, an endpoint MUST NOT initiate a key update while there are any unacknowledged handshake messages, see Section 6.2.

6.2. Key Update

Once the TLS handshake is complete, the KEY_PHASE bit allows for refreshes of keying material by either peer. Endpoints start using updated keys immediately without additional signaling; the change in the KEY_PHASE bit indicates that a new key is in use.

An endpoint MUST NOT initiate more than one key update at a time. A new key cannot be used until the endpoint has received and successfully decrypted a packet with a matching KEY_PHASE. Note that when 0-RTT is attempted the value of the KEY_PHASE bit will be different on packets sent by either peer.

A receiving endpoint detects an update when the KEY_PHASE bit doesn't match what it is expecting. It creates a new secret (see

Section 5.3) and the corresponding read key and IV. If the packet can be decrypted and authenticated using these values, then the keys it uses for packet protection are also updated. The next packet sent by the endpoint will then use the new keys.

An endpoint doesn't need to send packets immediately when it detects that its peer has updated keys. The next packet that it sends will simply use the new keys. If an endpoint detects a second update before it has sent any packets with updated keys it indicates that its peer has updated keys twice without awaiting a reciprocal update. An endpoint MUST treat consecutive key updates as a fatal error and abort the connection.

An endpoint SHOULD retain old keys for a short period to allow it to decrypt packets with smaller packet numbers than the packet that triggered the key update. This allows an endpoint to consume packets that are reordered around the transition between keys. Packets with higher packet numbers always use the updated keys and MUST NOT be decrypted with old keys.

Keys and their corresponding secrets SHOULD be discarded when an endpoint has received all packets with packet numbers lower than the lowest packet number used for the new key. An endpoint might discard keys if it determines that the length of the delay to affected packets is excessive.

This ensures that once the handshake is complete, packets with the same KEY_PHASE will have the same packet protection keys, unless there are multiple key updates in a short time frame succession and significant packet reordering.

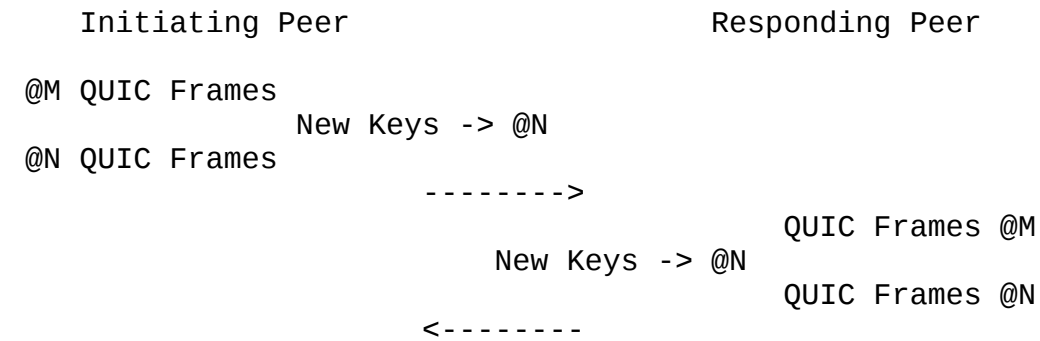

Figure 5: Key Update

As shown in Figure 3 and Figure 5, there is never a situation where there are more than two different sets of keying material that might be received by a peer. Once both sending and receiving keys have been updated, the peers immediately begin to use them.

A server cannot initiate a key update until it has received the client's Finished message. Otherwise, packets protected by the updated keys could be confused for retransmissions of handshake messages. A client cannot initiate a key update until all of its handshake messages have been acknowledged by the server.

A packet that triggers a key update could arrive after successfully processing a packet with a higher packet number. This is only possible if there is a key compromise and an attack, or if the peer is incorrectly reverting to use of old keys. Because the latter cannot be differentiated from an attack, an endpoint MUST immediately terminate the connection if it detects this condition.

7. Client Address Validation

Two tools are provided by TLS to enable validation of client source addresses at a server: the cookie in the HelloRetryRequest message, and the ticket in the NewSessionTicket message.

7.1. HelloRetryRequest Address Validation

The cookie extension in the TLS HelloRetryRequest message allows a server to perform source address validation during the handshake.

When QUIC requests address validation during the processing of the first ClientHello, the token it provides is included in the cookie extension of a HelloRetryRequest. As long as the cookie cannot be successfully guessed by a client, the server can be assured that the client received the HelloRetryRequest if it includes the value in a second ClientHello.

An initial ClientHello never includes a cookie extension. Thus, if a server constructs a cookie that contains all the information necessary to reconstruct state, it can discard local state after sending a HelloRetryRequest. Presence of a valid cookie in a ClientHello indicates that the ClientHello is a second attempt from the client.

An address validation token can be extracted from a second ClientHello and passed to the transport for further validation. If that validation fails, the server MUST fail the TLS handshake and send an illegal_parameter alert.

Combining address validation with the other uses of HelloRetryRequest ensures that there are fewer ways in which an additional round-trip can be added to the handshake. In particular, this makes it possible to combine a request for address validation with a request for a different client key share.

If TLS needs to send a HelloRetryRequest for other reasons, it needs to ensure that it can correctly identify the reason that the HelloRetryRequest was generated. During the processing of a second ClientHello, TLS does not need to consult the transport protocol regarding address validation if address validation was not requested originally. In such cases, the cookie extension could either be absent or it could indicate that an address validation token is not present.

7.1.1. Stateless Address Validation

A server can use the cookie extension to store all state necessary to continue the connection. This allows a server to avoid committing state for clients that have unvalidated source addresses.

For instance, a server could use a statically-configured key to encrypt the information that it requires and include that information in the cookie. In addition to address validation information, a server that uses encryption also needs to be able recover the hash of the ClientHello and its length, plus any information it needs in order to reconstruct the HelloRetryRequest.

7.1.2. Sending HelloRetryRequest

A server does not need to maintain state for the connection when sending a HelloRetryRequest message. This might be necessary to avoid creating a denial of service exposure for the server. However, this means that information about the transport will be lost at the server. This includes the stream offset of stream 0, the packet number that the server selects, and any opportunity to measure round trip time.

A server MUST send a TLS HelloRetryRequest in a Retry packet. Using a Retry packet causes the client to reset stream offsets. It also avoids the need for the server select an initial packet number, which would need to be remembered so that subsequent packets could be correctly numbered.

A HelloRetryRequest message MUST NOT be split between multiple Retry packets. This means that HelloRetryRequest is subject to the same size constraints as a ClientHello (see Section 4.4).

A client might send multiple Initial packets in response to loss. If a server sends a Retry packet in response to an Initial packet, it does not have to generate the same Retry packet each time. Variations in Retry packet, if used by a client, could lead to multiple connections derived from the same ClientHello. Reuse of the client nonce is not supported by TLS and could lead to security

vulnerabilities. Clients that receive multiple Retry packets MUST use only one and discard the remainder.

7.2. NewSessionTicket Address Validation

The ticket in the TLS NewSessionTicket message allows a server to provide a client with a similar sort of token. When a client resumes a TLS connection - whether or not 0-RTT is attempted - it includes the ticket in the handshake message. As with the HelloRetryRequest cookie, the server includes the address validation token in the ticket. TLS provides the token it extracts from the session ticket to the transport when it asks whether source address validation is needed.

If both a HelloRetryRequest cookie and a session ticket are present in the ClientHello, only the token from the cookie is passed to the transport. The presence of a cookie indicates that this is a second ClientHello - the token from the session ticket will have been provided to the transport when it appeared in the first ClientHello.

A server can send a NewSessionTicket message at any time. This allows it to update the state - and the address validation token - that is included in the ticket. This might be done to refresh the ticket or token, or it might be generated in response to changes in the state of the connection. QUIC can request that a NewSessionTicket be sent by providing a new address validation token.

A server that intends to support 0-RTT SHOULD provide an address validation token immediately after completing the TLS handshake.

7.3. Address Validation Token Integrity

TLS MUST provide integrity protection for address validation token unless the transport guarantees integrity protection by other means. For a NewSessionTicket that includes confidential information - such as the resumption secret - including the token under authenticated encryption ensures that the token gains both confidentiality and integrity protection without duplicating the overheads of that protection.

8. Pre-handshake QUIC Messages

Implementations MUST NOT exchange data on any stream other than stream 0 without packet protection. QUIC requires the use of several types of frame for managing loss detection and recovery during this phase. In addition, it might be useful to use the data acquired during the exchange of unauthenticated messages for congestion control.

This section generally only applies to TLS handshake messages from both peers and acknowledgments of the packets carrying those messages. In many cases, the need for servers to provide acknowledgments is minimal, since the messages that clients send are small and implicitly acknowledged by the server's responses.

The actions that a peer takes as a result of receiving an unauthenticated packet needs to be limited. In particular, state established by these packets cannot be retained once record protection commences.

There are several approaches possible for dealing with unauthenticated packets prior to handshake completion:

- discard and ignore them
- use them, but reset any state that is established once the handshake completes
- use them and authenticate them afterwards; failing the handshake if they can't be authenticated
- save them and use them when they can be properly authenticated
- treat them as a fatal error

Different strategies are appropriate for different types of data. This document proposes that all strategies are possible depending on the type of message.

- Transport parameters are made usable and authenticated as part of the TLS handshake (see Section 9.2).
- Most unprotected messages are treated as fatal errors when received except for the small number necessary to permit the handshake to complete (see Section 8.1).
- Protected packets can either be discarded or saved and later used (see Section 8.3).

8.1. Unprotected Packets Prior to Handshake Completion

This section describes the handling of messages that are sent and received prior to the completion of the TLS handshake.

Sending and receiving unprotected messages is hazardous. Unless expressly permitted, receipt of an unprotected message of any kind MUST be treated as a fatal error.

8.1.1. STREAM Frames

"STREAM" frames for stream 0 are permitted. These carry the TLS handshake messages. Once 1-RTT keys are available, unprotected "STREAM" frames on stream 0 can be ignored.

Receiving unprotected "STREAM" frames for other streams MUST be treated as a fatal error.

8.1.2. ACK Frames

"ACK" frames are permitted prior to the handshake being complete. Information learned from "ACK" frames cannot be entirely relied upon, since an attacker is able to inject these packets. Timing and packet retransmission information from "ACK" frames is critical to the functioning of the protocol, but these frames might be spoofed or altered.

Endpoints MUST NOT use an "ACK" frame in an unprotected packet to acknowledge packets that were protected by 0-RTT or 1-RTT keys. An endpoint MUST treat receipt of an "ACK" frame in an unprotected packet that claims to acknowledge protected packets as a connection error of type OPTIMISTIC_ACK. An endpoint that can read protected data is always able to send protected data.

Note: 0-RTT data can be acknowledged by the server as it receives it, but any packets containing acknowledgments of 0-RTT data cannot have packet protection removed by the client until the TLS handshake is complete. The 1-RTT keys necessary to remove packet protection cannot be derived until the client receives all server handshake messages.

An endpoint SHOULD use data from "ACK" frames carried in unprotected packets or packets protected with 0-RTT keys only during the initial handshake. All "ACK" frames contained in unprotected packets that are received after successful receipt of a packet protected with 1-RTT keys MUST be discarded. An endpoint SHOULD therefore include acknowledgments for unprotected and any packets protected with 0-RTT keys until it sees an acknowledgment for a packet that is both protected with 1-RTT keys and contains an "ACK" frame.

8.1.3. Updates to Data and Stream Limits

"MAX_DATA", "MAX_STREAM_DATA", "BLOCKED", "STREAM_BLOCKED", and "MAX_STREAM_ID" frames MUST NOT be sent unprotected.

Though data is exchanged on stream 0, the initial flow control window on that stream is sufficiently large to allow the TLS handshake to

complete. This limits the maximum size of the TLS handshake and would prevent a server or client from using an abnormally large certificate chain.

Stream 0 is exempt from the connection-level flow control window.

Consequently, there is no need to signal being blocked on flow control.

Similarly, there is no need to increase the number of allowed streams until the handshake completes.

8.1.4. Handshake Failures

The "CONNECTION_CLOSE" frame MAY be sent by either endpoint in a Handshake packet. This allows an endpoint to signal a fatal error with connection establishment. A "STREAM" frame carrying a TLS alert MAY be included in the same packet.

8.1.5. Address Verification

In order to perform source-address verification before the handshake is complete, "PATH_CHALLENGE" and "PATH_RESPONSE" frames MAY be exchanged unprotected.

8.1.6. Denial of Service with Unprotected Packets

Accepting unprotected - specifically unauthenticated - packets presents a denial of service risk to endpoints. An attacker that is able to inject unprotected packets can cause a recipient to drop even protected packets with a matching packet number. The spurious packet shadows the genuine packet, causing the genuine packet to be ignored as redundant.

Once the TLS handshake is complete, both peers MUST ignore unprotected packets. From that point onward, unprotected messages can be safely dropped.

Since only TLS handshake packets and acknowledgments are sent in the clear, an attacker is able to force implementations to rely on retransmission for packets that are lost or shadowed. Thus, an attacker that intends to deny service to an endpoint has to drop or shadow protected packets in order to ensure that their victim continues to accept unprotected packets. The ability to shadow packets means that an attacker does not need to be on path.

In addition to causing valid packets to be dropped, an attacker can generate packets with an intent of causing the recipient to expend

processing resources. See Section 10.2 for a discussion of these risks.

To avoid receiving TLS packets that contain no useful data, a TLS implementation MUST reject empty TLS handshake records and any record that is not permitted by the TLS state machine. Any TLS application data or alerts that are received prior to the end of the handshake MUST be treated as a connection error of type PROTOCOL_VIOLATION.

8.2. Use of 0-RTT Keys

If 0-RTT keys are available (see Section 5.2), the lack of replay protection means that restrictions on their use are necessary to avoid replay attacks on the protocol.

A client MUST only use 0-RTT keys to protect data that is idempotent. A client MAY wish to apply additional restrictions on what data it sends prior to the completion of the TLS handshake. A client otherwise treats 0-RTT keys as equivalent to 1-RTT keys.

A client that receives an indication that its 0-RTT data has been accepted by a server can send 0-RTT data until it receives all of the server's handshake messages. A client SHOULD stop sending 0-RTT data if it receives an indication that 0-RTT data has been rejected.

A server MUST NOT use 0-RTT keys to protect packets.

If a server rejects 0-RTT, then the TLS stream will not include any TLS records protected with 0-RTT keys.

8.3. Receiving Out-of-Order Protected Frames

Due to reordering and loss, protected packets might be received by an endpoint before the final TLS handshake messages are received. A client will be unable to decrypt 1-RTT packets from the server, whereas a server will be able to decrypt 1-RTT packets from the client.

Packets protected with 1-RTT keys MAY be stored and later decrypted and used once the handshake is complete. A server MUST NOT use 1-RTT protected packets before verifying either the client Finished message or - in the case that the server has chosen to use a pre-shared key - the pre-shared key binder (see Section 4.2.8 of [TLS13]). Verifying these values provides the server with an assurance that the ClientHello has not been modified.

A server could receive packets protected with 0-RTT keys prior to receiving a TLS ClientHello. The server MAY retain these packets for later decryption in anticipation of receiving a ClientHello.

Receiving and verifying the TLS Finished message is critical in ensuring the integrity of the TLS handshake. A server MUST NOT use protected packets from the client prior to verifying the client Finished message if its response depends on client authentication.

9. QUIC-Specific Additions to the TLS Handshake

QUIC uses the TLS handshake for more than just negotiation of cryptographic parameters. The TLS handshake validates protocol version selection, provides preliminary values for QUIC transport parameters, and allows a server to perform return routeability checks on clients.

9.1. Protocol and Version Negotiation

The QUIC version negotiation mechanism is used to negotiate the version of QUIC that is used prior to the completion of the handshake. However, this packet is not authenticated, enabling an active attacker to force a version downgrade.

To ensure that a QUIC version downgrade is not forced by an attacker, version information is copied into the TLS handshake, which provides integrity protection for the QUIC negotiation. This does not prevent version downgrade prior to the completion of the handshake, though it means that a downgrade causes a handshake failure.

TLS uses Application Layer Protocol Negotiation (ALPN) [RFC7301] to select an application protocol. The application-layer protocol MAY restrict the QUIC versions that it can operate over. Servers MUST select an application protocol compatible with the QUIC version that the client has selected.

If the server cannot select a compatible combination of application protocol and QUIC version, it MUST abort the connection. A client MUST abort a connection if the server picks an incompatible combination of QUIC version and ALPN identifier.

9.2. QUIC Transport Parameters Extension

QUIC transport parameters are carried in a TLS extension. Different versions of QUIC might define a different format for this struct.

Including transport parameters in the TLS handshake provides integrity protection for these values.

```
enum {
   quic_transport_parameters(26), (65535)
} ExtensionType;
```

The "extension_data" field of the quic_transport_parameters extension contains a value that is defined by the version of QUIC that is in use. The quic_transport_parameters extension carries a TransportParameters when the version of QUIC defined in [QUIC-TRANSPORT] is used.

The quic_transport_parameters extension is carried in the ClientHello and the EncryptedExtensions messages during the handshake.

10. Security Considerations

There are likely to be some real clangers here eventually, but the current set of issues is well captured in the relevant sections of the main text.

Never assume that because it isn't in the security considerations section it doesn't affect security. Most of this document does.

10.1. Packet Reflection Attack Mitigation

A small ClientHello that results in a large block of handshake messages from a server can be used in packet reflection attacks to amplify the traffic generated by an attacker.

Certificate caching [RFC7924] can reduce the size of the server's handshake messages significantly.

QUIC requires that the packet containing a ClientHello be padded to a minimum size. A server is less likely to generate a packet reflection attack if the data it sends is a small multiple of this size. A server SHOULD use a HelloRetryRequest if the size of the handshake messages it sends is likely to significantly exceed the size of the packet containing the ClientHello.

10.2. Peer Denial of Service

QUIC, TLS and HTTP/2 all contain a messages that have legitimate uses in some contexts, but that can be abused to cause a peer to expend processing resources without having any observable impact on the state of the connection. If processing is disproportionately large in comparison to the observable effects on bandwidth or state, then this could allow a malicious peer to exhaust processing capacity without consequence.

QUIC prohibits the sending of empty "STREAM" frames unless they are marked with the FIN bit. This prevents "STREAM" frames from being sent that only waste effort.

TLS records SHOULD always contain at least one octet of a handshake messages or alert. Records containing only padding are permitted during the handshake, but an excessive number might be used to generate unnecessary work. Once the TLS handshake is complete, endpoints MUST NOT send TLS application data records. Receiving TLS application data MUST be treated as a connection error of type PROTOCOL_VIOLATION.

While there are legitimate uses for some redundant packets, implementations SHOULD track redundant packets and treat excessive volumes of any non-productive packets as indicative of an attack.

10.3. Packet Number Protection Analysis

Packet number protection relies the packet protection AEAD being a pseudorandom function (PRF), which is not a property that AEAD algorithms guarantee. Therefore, no strong assurances about the general security of this mechanism can be shown in the general case. The AEAD algorithms described in this document are assumed to be PRFs.

The packet number protection algorithms defined in this document take the form:

"encrypted_pn = packet_number XOR PRF(pn_key, sample) "

This construction is secure against chosen plaintext attacks (IND-CPA) [IMC].

Use of the same key and ciphertext sample more than once risks compromising packet number protection. Protecting two different packet numbers with the same key and ciphertext sample reveals the exclusive OR of those packet numbers. Assuming that the AEAD acts as a PRF, if L bits are sampled, the odds of two ciphertext samples being identical approach 2^(-L/2), that is, the birthday bound. For the algorithms described in this document, that probability is one in 2^64.

Note: In some cases, inputs shorter than the full size required by the packet protection algorithm might be used.

To prevent an attacker from modifying packet numbers, values of packet numbers are transitively authenticated using packet protection; packet numbers are part of the authenticated additional

data. A falsified or modified packet number can only be detected once the packet protection is removed.

An attacker can guess values for packet numbers and have an endpoint confirm guesses through timing side channels. If the recipient of a packet discards packets with duplicate packet numbers without attempting to remove packet protection they could reveal through timing side-channels that the packet number matches a received packet. For authentication to be free from side-channels, the entire process of packet number protection removal, packet number recovery, and packet protection removal MUST be applied together without timing and other side-channels.

For the sending of packets, construction and protection of packet payloads and packet numbers MUST be free from side-channels that would reveal the packet number or its encoded size.

11. Error Codes

This section defines error codes from the error code space used in [QUIC-TRANSPORT].

The following error codes are defined when TLS is used for the crypto handshake:

TLS_HANDSHAKE_FAILED (0x201): The TLS handshake failed.

TLS_FATAL_ALERT_GENERATED (0x202): A TLS fatal alert was sent, causing the TLS connection to end prematurely.

TLS_FATAL_ALERT_RECEIVED (0x203): A TLS fatal alert was received, causing the TLS connection to end prematurely.

12. IANA Considerations

This document does not create any new IANA registries, but it registers the values in the following registries:

- QUIC Transport Error Codes Registry [QUIC-TRANSPORT] - IANA is to register the three error codes found in Section 11, these are summarized in Table 1.

- TLS ExtensionsType Registry [TLS-REGISTRIES] - IANA is to register the quic_transport_parameters extension found in Section 9.2. Assigning 26 to the extension would be greatly appreciated. The Recommended column is to be marked Yes. The TLS 1.3 Column is to include CH and EE.

- TLS Exporter Label Registry [TLS-REGISTRIES] - IANA is requested to register "EXPORTER-QUIC 0rtt" from Section 5.3.3; "EXPORTER-QUIC client 1rtt" and "EXPORTER-QUIC server 1-RTT" from Section 5.3.4. The DTLS column is to be marked No. The Recommended column is to be marked Yes.

Value	Error	Description	Specification
0x201	TLS_HANDSHAKE_FAILED	TLS handshake failure	Section 11
0x202	TLS_FATAL_ALERT_GENERATED	Sent TLS alert	Section 11
0x203	TLS_FATAL_ALERT_RECEIVED	Receives TLS alert	Section 11

Table 1: QUIC Transport Error Codes for TLS

13. References

13.1. Normative References

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

[AES] "Advanced encryption standard (AES)", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.197, November 2001.

[CHACHA] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 7539, DOI 10.17487/RFC7539, May 2015, <https://www.rfc-editor.org/info/rfc7539>.

[HKDF] 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/info/rfc5869>.

[QUIC-TRANSPORT] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-12 (work in progress), May 2018.

[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/info/rfc2119>.

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

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/info/rfc7301>.

[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/info/rfc8174>.

[SHA] Dang, Q., "Secure Hash Standard", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.180-4, July 2015.

[TLS-REGISTRIES] Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", draft-ietf-tls-iana-registry-updates-04 (work in progress), February 2018.

[TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-21 (work in progress), July 2017.

13.2. Informative References

[AEBounds] Luykx, A. and K. Paterson, "Limits on Authenticated Encryption Use in TLS", March 2016, <http://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf>.

[IMC] Katz, J. and Y. Lindell, "Introduction to Modern Cryptography, Second Edition", ISBN 978-1466570269, November 2014.

[QUIC-HTTP] Bishop, M., Ed., "Hypertext Transfer Protocol (HTTP) over QUIC", draft-ietf-quic-http-12 (work in progress), May 2018.

[QUIC-RECOVERY]
Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control", draft-ietf-quic-recovery-11 (work in progress), May 2018.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <https://www.rfc-editor.org/info/rfc2818>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

[RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, July 2016, <https://www.rfc-editor.org/info/rfc7924>.

13.3. URIs

[1] https://mailarchive.ietf.org/arch/search/?email_list=quic

[2] https://github.com/quicwg

[3] https://github.com/quicwg/base-drafts/labels/-tls

Appendix A. Contributors

Ryan Hamilton was originally an author of this specification.

Appendix B. Acknowledgments

This document has benefited from input from Dragana Damjanovic, Christian Huitema, Jana Iyengar, Adam Langley, Roberto Peon, Eric Rescorla, Ian Swett, and many others.

Appendix C. Change Log

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

Issue and pull request numbers are listed with a leading octothorp.

C.1. Since draft-ietf-quic-tls-10

- No significant changes.

C.2. Since draft-ietf-quic-tls-09

- Cleaned up key schedule and updated the salt used for handshake packet protection (#1077)

C.3. Since draft-ietf-quic-tls-08

- Specify value for max_early_data_size to enable 0-RTT (#942)
- Update key derivation function (#1003, #1004)

C.4. Since draft-ietf-quic-tls-07

- Handshake errors can be reported with CONNECTION_CLOSE (#608, #891)

C.5. Since draft-ietf-quic-tls-05

No significant changes.

C.6. Since draft-ietf-quic-tls-04

- Update labels used in HKDF-Expand-Label to match TLS 1.3 (#642)

C.7. Since draft-ietf-quic-tls-03

No significant changes.

C.8. Since draft-ietf-quic-tls-02

- Updates to match changes in transport draft

C.9. Since draft-ietf-quic-tls-01

- Use TLS alerts to signal TLS errors (#272, #374)
- Require ClientHello to fit in a single packet (#338)
- The second client handshake flight is now sent in the clear (#262, #337)
- The QUIC header is included as AEAD Associated Data (#226, #243, #302)

- Add interface necessary for client address validation (#275)
- Define peer authentication (#140)
- Require at least TLS 1.3 (#138)
- Define transport parameters as a TLS extension (#122)
- Define handling for protected packets before the handshake completes (#39)
- Decouple QUIC version and ALPN (#12)

C.10. Since draft-ietf-quic-tls-00

- Changed bit used to signal key phase
- Updated key phase markings during the handshake
- Added TLS interface requirements section
- Moved to use of TLS exporters for key derivation
- Moved TLS error code definitions into this document

C.11. Since draft-thomson-quic-tls-01

- Adopted as base for draft-ietf-quic-tls
- Updated authors/editors list
- Added status note

Authors' Addresses

Martin Thomson (editor)
Mozilla

Email: martin.thomson@gmail.com

Sean Turner (editor)
sn3rd

Email: sean@sn3rd.com