QUIC
Internet-Draft
Intended status: Standards Track
Expires: July 18, 2017

M. Thomson, Ed.
Mozilla
S. Turner, Ed.
sn3rd

January 14, 2017

Using Transport Layer Security (TLS) to Secure QUIC
draft-ietf-quic-tls-01

Abstract

This document describes how Transport Layer Security (TLS) can be 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 .

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

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 http://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 July 18, 2017.

Copyright Notice

Table of Contents

1. Introduction

QUIC [QUIC-TRANSPORT] provides a multiplexed transport. When used for HTTP [RFC7230] semantics [QUIC-HTTP] it provides several key advantages over HTTP/1.1 [RFC7230] or HTTP/2 [RFC7540] over TCP [RFC0793].

This document describes how QUIC can be secured using Transport Layer Security (TLS) version 1.3 [I-D.ietf-tls-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, zero round trip setup.

This document describes how the standardized TLS 1.3 can act 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 words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It's not shouting; when they are capitalized, they have the special meaning defined in [RFC2119].

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 [I-D.ietf-tls-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.

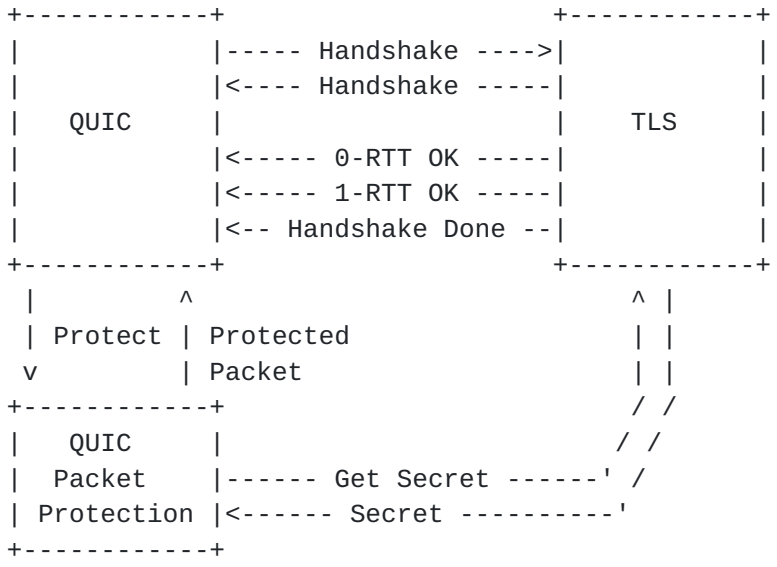

Figure 1: QUIC and TLS Interactions

The initial state of a QUIC connection has packets exchanged without any form of protection. In this state, QUIC is limited to using stream 1 and associated packets. Stream 1 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.

This arrangement means that some TLS messages receive redundant protection from both the QUIC packet protection and the TLS record protection. These messages are limited in number; the TLS connection is rarely needed once the handshake completes.

3.1. TLS Overview

TLS provides two endpoints 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; QUIC 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 exchange. 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 certificate-based authentication [RFC5280] 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 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 immediately. This 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 [I-D.ietf-tls-tls13] for more options and details.

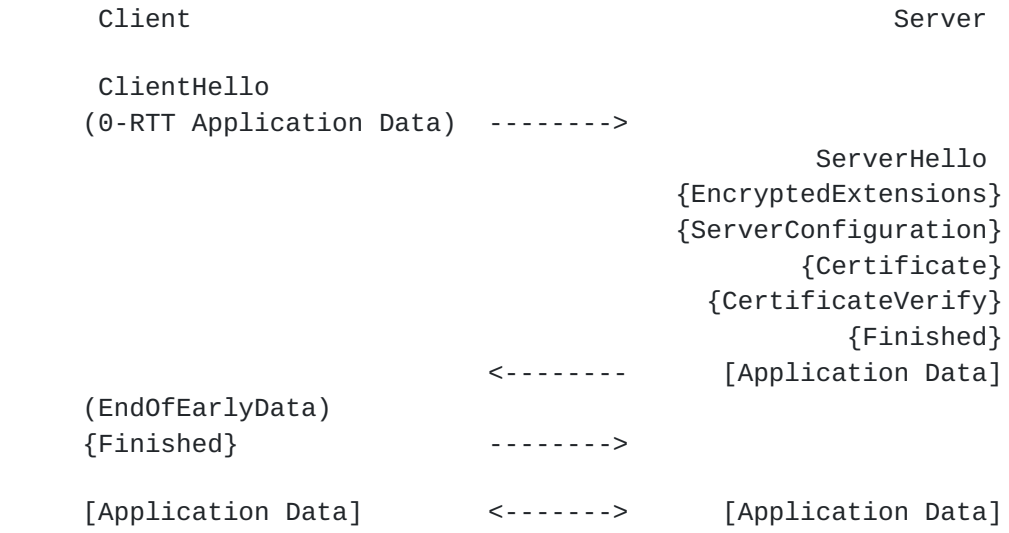

Figure 2: TLS Handshake with 0-RTT

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 Section 8.3).

- A pre-shared key mode can be used for subsequent handshakes to avoid 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 1 for a TLS connection. Stream 1 contains a complete TLS connection, which includes the TLS record layer. Other than the definition of a QUIC-specific extension (see Section-TBD), 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 1 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 7.3.3 of [I-D.ietf-tls-tls13] and Section 5.2). 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 1 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.

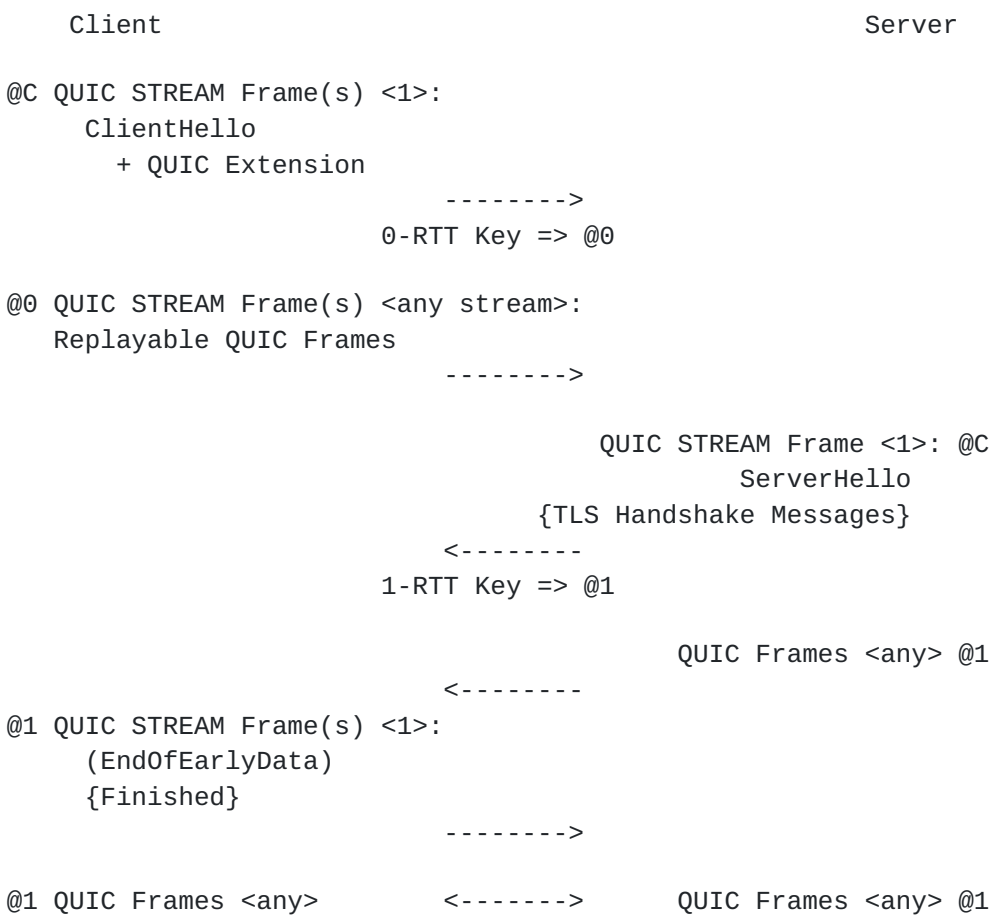

Figure 3: QUIC over TLS Handshake

In Figure 3, symbols mean:

- "<" and ">" enclose stream numbers.
- "@" indicates the key phase that is currently used for protecting QUIC packets.
- "(" 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 unprotected packets (@C) to 1-RTT protection (@1), which happens before it sends its final set of TLS handshake messages.

The server sends TLS handshake messages without protection (@C). The server transitions from no protection (@C) 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 cleartext (@C) to 0-RTT keys (@0) when sending 0-RTT data, and subsequently to to 1-RTT keys (@1) for 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 three primary functions: Handshake, 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 1. There are two basic functions on this interface: one where QUIC requests handshake messages and one where QUIC provides handshake packets.

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 1 octets.

Each time that an endpoint receives data on stream 1, 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.

Once the TLS handshake is complete, this is indicated to QUIC along with any final handshake octets that TLS needs to send. Once the handshake is complete, TLS becomes passive. TLS can still receive data from its peer and respond in kind that data, 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 1. 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.

4.2.2. 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 has enough information to generate 1-RTT keys, it indicates their availability. On the client, this occurs after receiving the entirety of the first flight of TLS handshake messages from the server. A server indicates that 1-RTT keys are available after it sends its handshake messages.

This ordering ensures that a client sends its second flight of handshake messages protected with 1-RTT keys. More importantly, it ensures that the server sends its flight of handshake messages without protection.

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 both sending and receiving packets. 0-RTT keys are only used to protect packets that the client sends.

4.2.3. Secret Export

Details how secrets are exported from TLS are included in Section 5.2.

4.2.4. TLS Interface Summary

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

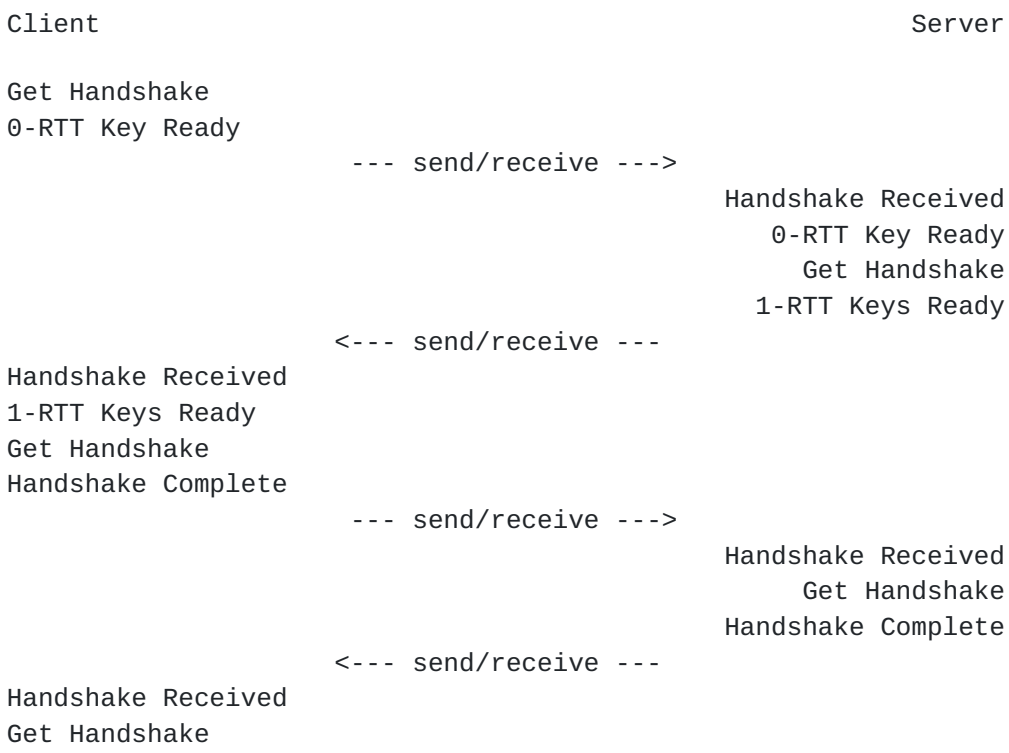

Figure 4: Interaction Summary between QUIC and TLS

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 5.3). Packet protection uses keys that are exported from the TLS connection (see Section 5.2).

Different keys are used for QUIC packet protection and TLS record protection. Having separate QUIC and TLS record protection means that TLS records can be protected by two different keys. This redundancy is limited to a only a few TLS records, and is maintained for the sake of simplicity.

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 5.2). The selection of AEAD function is also updated to match the AEAD negotiated by TLS.

For packets other than any unprotected handshake packets (see Section 6.1), once a change of keys has been made, packets with higher packet numbers MUST use 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. QUIC Key Expansion

QUIC uses a system of packet protection secrets, keys and IVs that are modelled on the system used in TLS [I-D.ietf-tls-tls13]. The secrets that QUIC uses as the basis of its key schedule are obtained using TLS exporters (see Section 7.3.3 of [I-D.ietf-tls-tls13]).

QUIC uses the Pseudo-Random Function (PRF) 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.

5.2.1. 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 7.2. 0-RTT keys are used after sending or receiving a ClientHello.

The secret is exported from TLS using the exporter label "EXPORTER-QUIC 0-RTT Secret" 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-Exporter("EXPORTER-QUIC 0-RTT Secret"
                   "", Hash.length)
```

5.2.2. 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 protecting packets sent by the server.

The initial client packet protection secret is exported from TLS using the exporter label "EXPORTER-QUIC client 1-RTT Secret"; the initial server packet protection secret uses the exporter label "EXPORTER-QUIC server 1-RTT Secret". 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 1-RTT Secret"
                   "", Hash.length)
server_pp_secret_0
    = TLS-Exporter("EXPORTER-QUIC server 1-RTT Secret"
                   "", Hash.length)
```

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

After a key update (see Section 6.2), these secrets are updated using the HKDF-Expand-Label function defined in Section 7.1 of [I-D.ietf-tls-tls13]. HKDF-Expand-Label uses the the PRF hash function negotiated by TLS. The replacement secret is derived using the existing Secret, a Label of "QUIC client 1-RTT Secret" for the client and "QUIC server 1-RTT Secret" for the server, an empty HashValue, and the same output Length as the hash function selected by TLS for its PRF.

```
client_pp_secret_<N+1>
    = HKDF-Expand-Label(client_pp_secret_<N>,
                        "QUIC client 1-RTT Secret",
                        "", Hash.length)
server_pp_secret_<N+1>
    = HKDF-Expand-Label(server_pp_secret_<N>,
                        "QUIC server 1-RTT Secret",
                        "", Hash.length)
```

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

HKDF-Expand-Label uses HKDF-Expand [RFC5869] with a specially formatted info parameter. The info parameter that includes the output length (in this case, the size of the PRF hash output) encoded on two octets in network byte order, the length of the prefixed Label

as a single octet, the value of the Label prefixed with "TLS 1.3, ", and a zero octet to indicate an empty HashValue. For example, the client packet protection secret uses an info parameter of:

```
info = (HashLen / 256) || (HashLen % 256) || 0x21 ||
       "TLS 1.3, QUIC client 1-RTT secret" || 0x00
```

5.2.3. Packet Protection Key and IV

The complete key expansion uses an identical process for key expansion as defined in Section 7.3 of [I-D.ietf-tls-tls13], using different values for the input secret. 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 use the QUIC 0-RTT secret. This uses the HKDF-Expand-Label with the PRF hash function negotiated by TLS.

The length of the 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 [I-D.ietf-tls-tls13], the IV length is the larger of 8 or N_MIN (see Section 4 of [RFC5116]).

```
client_0rtt_key = HKDF-Expand-Label(client_0rtt_secret,
                                    "key", "", key_length)
client_0rtt_iv = HKDF-Expand-Label(client_0rtt_secret,
                                   "iv", "", iv_length)
```

Similarly, the packet protection key and IV used to protect 1-RTT packets sent by both client and server use the current packet protection secret.

```
client_pp_key_<N> = HKDF-Expand-Label(client_pp_secret_<N>,
                                      "key", "", key_length)
client_pp_iv_<N> = HKDF-Expand-Label(client_pp_secret_<N>,
                                     "iv", "", iv_length)
server_pp_key_<N> = HKDF-Expand-Label(server_pp_secret_<N>,
                                      "key", "", key_length)
server_pp_iv_<N> = HKDF-Expand-Label(server_pp_secret_<N>,
                                     "iv", "", iv_length)
```

The client protects (or encrypts) packets with the client packet protection key and IV; the server protects packets with the server packet protection key.

The QUIC record protection initially starts without keying material. When the TLS state machine reports that the ClientHello has been sent, the 0-RTT keys can be generated and installed for writing.

When the TLS state machine reports completion of the handshake, the 1-RTT keys can be generated and installed for writing.

5.3. QUIC AEAD Usage

The Authentication Encryption with Associated Data (AEAD) [RFC5116] 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.

Regular QUIC packets are protected by an AEAD [RFC5116]. Version negotiation and public reset packets are not protected.

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, for the AEAD is either the client packet protection key (client_pp_key_n) or the server packet protection key (server_pp_key_n), derived as defined in Section 5.2.

The nonce, N, for the AEAD is formed by combining either the packet protection IV (either client_pp_iv_n or server_pp_iv_n) with packet numbers. 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 an empty sequence.

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

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

Prior to TLS providing keys, no record protection is performed and the plaintext, P, is transmitted unmodified.

5.4. 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 packet number is not reset and it is not permitted to go higher than its maximum value of 2^64-1. 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 1. This sequence number is not visible to QUIC.

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. The exception is the transition from 0-RTT keys to 1-RTT keys, where the presence of the version field and its associated bit is used (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 the third bit of the public flags (0x04).

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 are sent without protection. These packets are marked with a KEY_PHASE of 0.

TLS handshake messages that are critical to the TLS key exchange cannot be protected using QUIC packet protection. A KEY_PHASE of 0 is used for all of these packets, even during retransmission. The messages critical to key exchange are the TLS ClientHello and any TLS handshake message from the server, except those that are sent after the handshake completes, such as NewSessionTicket.

The second flight of TLS handshake messages from the client, and any TLS handshake messages that are sent after completing the TLS handshake do not need special packet protection rules. This includes the EndOfEarlyData message that is sent by a client to mark the end of its 0-RTT data. 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 unprotected packets (KEY_PHASE=0).

6.1.1. Initial Key Transitions

Once the TLS key exchange is complete, keying material is exported from TLS and QUIC packet protection commences.

Packets protected with 1-RTT keys have a KEY_PHASE bit set to 1. These packets also have a VERSION bit set to 0.

If the client is unable to send 0-RTT data - or it does not have 0-RTT data to send - packet protection with 1-RTT keys starts with the packets that contain its second flight of TLS handshake messages. That is, the flight containing the TLS Finished handshake message and optionally a Certificate and CertificateVerify message.

If the client sends 0-RTT data, it marks packets protected with 0-RTT keys with a KEY_PHASE of 1 and a VERSION bit of 1. Setting the version bit means that all packets also include the version field. The client removes the VERSION bit when it transitions to using 1-RTT keys, but it does not change the KEY_PHASE bit.

Marking 0-RTT data with the both KEY_PHASE and VERSION bits ensures that the server is able to identify these packets as 0-RTT data in case the packet containing the TLS ClientHello is lost or delayed.

Including the version also ensures that the packet format is known to the server in this case.

Using both KEY_PHASE and VERSION also ensures that the server is able to distinguish between cleartext handshake packets (KEY_PHASE=0, VERSION=1), 0-RTT protected packets (KEY_PHASE=1, VERSION=1), and 1-RTT protected packets (KEY_PHASE=1, VERSION=0). Packets with all of these markings can arrive concurrently, and being able to identify each cleanly ensures that the correct packet protection keys can be selected and applied.

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. From this point, the server protects all packets with 1-RTT keys. Future packets are therefore protected with 1-RTT keys and marked with a KEY_PHASE of 1.

6.1.2. Retransmission and Acknowledgment of Unprotected Packets

The first flight of TLS handshake messages from both client and server (ClientHello, or ServerHello through to the server's Finished) are critical to the key exchange. The contents of these messages determines 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 retranmitting, retransmissions of these handshake messages MUST be sent in unprotected packets (with a KEY_PHASE of 0). An endpoint MUST also generate ACK frames for these messages that are sent in unprotected packets.

The TLS handshake messages that are affected by this rule are specifically:

- A client MUST NOT restransmit a TLS ClientHello with 0-RTT keys. The server needs this message in order to determine the 0-RTT keys.

- A server MUST NOT retransmit any of its TLS handshake messages with 1-RTT keys. The client needs these messages in order to determine the 1-RTT keys.

A HelloRetryRequest handshake message might be used to reject an initial ClientHello. A HelloRetryRequest handshake message and any second ClientHello that is sent in response MUST also be sent without packet protection. This is natural, because no new keying material 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.

Note: TLS handshake data that needs to be sent without protection is all the handshake data acquired from TLS before the point that 1-RTT keys are provided by TLS (see Section 4.2.2).

The KEY_PHASE and VERSION bits ensure 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.

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 5.2) 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 sequence numbers lower than the lowest sequence 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.

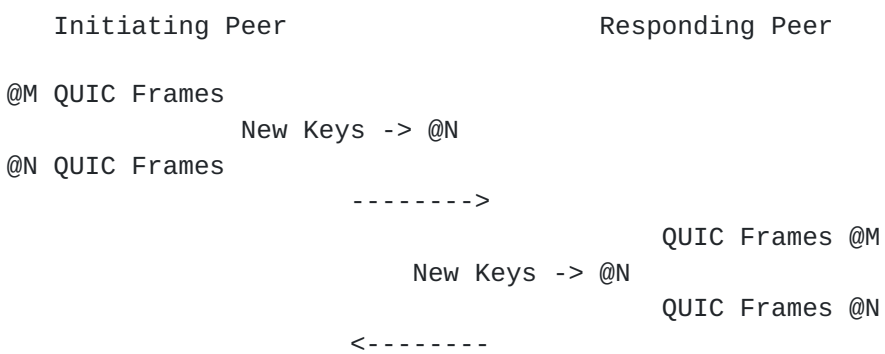

Figure 5: Key Update

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,

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.

7. Pre-handshake QUIC Messages

Implementations MUST NOT exchange data on any stream other than stream 1 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 and options are made usable and authenticated as part of the TLS handshake (see Section 8.2).

- Most unprotected messages are treated as fatal errors when received except for the small number necessary to permit the handshake to complete (see Section 7.1).

- Protected packets can either be discarded or saved and later used (see Section 7.3).

7.1. 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.

7.1.1. STREAM Frames

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

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

7.1.2. 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 unprotected "ACK" frame to acknowledge data that was protected by 0-RTT or 1-RTT keys. An endpoint MUST ignore an unprotected "ACK" frame if it claims to acknowledge data that was sent in a protected packet. Such an acknowledgement can only serve as a denial of service, since an endpoint that can read protected data is always able to send protected data.

ISSUE: What about 0-RTT data? Should we allow acknowledgment of 0-RTT with unprotected frames? If we don't, then 0-RTT data will be unacknowledged until the handshake completes. This isn't a problem if the handshake completes without loss, but it could mean

that 0-RTT stalls when a handshake packet disappears for any reason.

An endpoint SHOULD use data from unprotected or 0-RTT-protected "ACK" frames only during the initial handshake and while they have insufficient information from 1-RTT-protected "ACK" frames. Once sufficient information has been obtained from protected messages, information obtained from less reliable sources can be discarded.

7.1.3. WINDOW_UPDATE Frames

"WINDOW_UPDATE" frames MUST NOT be sent unprotected.

Though data is exchanged on stream 1, the initial flow control window is 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 1 is exempt from the connection-level flow control window.

7.1.4. 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 sequence 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.

ISSUE: This would not be an issue if QUIC had a randomized starting sequence number. If we choose to randomize, we fix this problem and reduce the denial of service exposure to on-path attackers. The only possible problem is in authenticating the initial value, so that peers can be sure that they haven't missed an initial message.

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 9.2 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 is received prior to the end of the handshake MUST be treated as a fatal error.

7.2. Use of 0-RTT Keys

If 0-RTT keys are available, 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.

7.3. Protected Packets Prior to Handshake Completion

Due to reordering and loss, protected packets might be received by an endpoint before the final handshake messages are received. If these can be decrypted successfully, such packets MAY be stored and used once the handshake is complete.

Unless expressly permitted below, encrypted packets MUST NOT be used prior to completing the TLS handshake, in particular the receipt of a valid Finished message and any authentication of the peer. If packets are processed prior to completion of the handshake, an attacker might use the willingness of an implementation to use these packets to mount attacks.

TLS handshake messages are covered by record protection during the handshake, once key agreement has completed. This means that protected messages need to be decrypted to determine if they are TLS handshake messages or not. Similarly, "ACK" and "WINDOW_UPDATE" frames might be needed to successfully complete the TLS handshake.

Any timestamps present in "ACK" frames MUST be ignored rather than causing a fatal error. Timestamps on protected frames MAY be saved and used once the TLS handshake completes successfully.

An endpoint MAY save the last protected "WINDOW_UPDATE" frame it receives for each stream and apply the values once the TLS handshake completes. Failing to do this might result in temporary stalling of affected streams.

8. 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.

8.1. 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 during the handshake, though it means that such a downgrade causes a handshake failure.

Protocols that use the QUIC transport MUST use Application Layer Protocol Negotiation (ALPN) [RFC7301]. The ALPN identifier for the protocol MUST be specific to the QUIC version that it operates over. When constructing a ClientHello, clients MUST include a list of all the ALPN identifiers that they support, regardless of whether the QUIC version that they have currently selected supports that protocol.

Servers SHOULD select an application protocol based solely on the information in the ClientHello, not using the QUIC version that the client has selected. If the protocol that is selected is not supported with the QUIC version that is in use, the server MAY send a QUIC version negotiation packet to select a compatible version.

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

8.2. QUIC Extension

QUIC defines an extension for use with TLS. That extension defines transport-related parameters. This provides integrity protection for these values. Including these in the TLS handshake also make the values that a client sets available to a server one-round trip earlier than parameters that are carried in QUIC packets. This document does not define that extension.

8.3. Source Address Validation

QUIC implementations describe a source address token. This is an opaque blob that a server might provide to clients when they first use a given source address. The client returns this token in subsequent messages as a return routeability check. That is, the client returns this token to prove that it is able to receive packets at the source address that it claims. This prevents the server from being used in packet reflection attacks (see Section 9.1).

A source address token is opaque and consumed only by the server. Therefore it can be included in the TLS 1.3 pre-shared key identifier for 0-RTT handshakes. Servers that use 0-RTT are advised to provide new pre-shared key identifiers after every handshake to avoid linkability of connections by passive observers. Clients MUST use a new pre-shared key identifier for every connection that they initiate; if no pre-shared key identifier is available, then resumption is not possible.

A server that is under load might include a source address token in the cookie extension of a HelloRetryRequest.

8.4. Priming 0-RTT

QUIC uses TLS without modification. Therefore, it is possible to use a pre-shared key that was obtained in a TLS connection over TCP to enable 0-RTT in QUIC. Similarly, QUIC can provide a pre-shared key that can be used to enable 0-RTT in TCP.

All the restrictions on the use of 0-RTT apply, with the exception of the ALPN label, which MUST only change to a label that is explicitly designated as being compatible. The client indicates which ALPN label it has chosen by placing that ALPN label first in the ALPN extension.

The certificate that the server uses MUST be considered valid for both connections, which will use different protocol stacks and could use different port numbers. For instance, HTTP/1.1 and HTTP/2 operate over TLS and TCP, whereas QUIC operates over UDP.

Source address validation is not completely portable between different protocol stacks. Even if the source IP address remains constant, the port number is likely to be different. Packet reflection attacks are still possible in this situation, though the set of hosts that can initiate these attacks is greatly reduced. A server might choose to avoid source address validation for such a connection, or allow an increase to the amount of data that it sends toward the client without source validation.

9. 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.

9.1. 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.

A client SHOULD also pad [RFC7685] its ClientHello to at least 1024 octets. A server is less likely to generate a packet reflection attack if the data it sends is a small multiple of the data it receives. A server SHOULD use a HelloRetryRequest if the size of the handshake messages it sends is likely to exceed the size of the ClientHello.

9.2. 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 SHOULD NOT send TLS application data records unless it is to hide the length of QUIC records. QUIC packet protection does not include any allowance for padding; padded TLS application data records can be used to mask the length of QUIC frames.

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. Error codes

The portion of the QUIC error code space allocated for the crypto handshake is 0xB000-0xFFFF. The following error codes are defined when TLS is used for the crypto handshake:

TLS_HANDSHAKE_FAILED (0xB01c): Crypto errors. Handshake failed.

TLS_MESSAGE_OUT_OF_ORDER (0xB01d): Handshake message received out of order.

TLS_TOO_MANY_ENTRIES (0xB01e): Handshake message contained too many entries.

TLS_INVALID_VALUE_LENGTH (0xB01f): Handshake message contained an invalid value length.

TLS_MESSAGE_AFTER_HANDSHAKE_COMPLETE (0xB020): A handshake message was received after the handshake was complete.

TLS_INVALID_RECORD_TYPE (0xB021): A handshake message was received with an illegal record type.

TLS_INVALID_PARAMETER (0xB022): A handshake message was received with an illegal parameter.

TLS_INVALID_CHANNEL_ID_SIGNATURE (0xB034): An invalid channel id signature was supplied.

TLS_MESSAGE_PARAMETER_NOT_FOUND (0xB023): A handshake message was received with a mandatory parameter missing.

TLS_MESSAGE_PARAMETER_NO_OVERLAP (0xB024): A handshake message was received with a parameter that has no overlap with the local parameter.

TLS_MESSAGE_INDEX_NOT_FOUND (0xB025): A handshake message was received that contained a parameter with too few values.

TLS_UNSUPPORTED_PROOF_DEMAND (0xB05e): A demand for an unsupported proof type was received.

TLS_INTERNAL_ERROR (0xB026): An internal error occured in handshake processing.

TLS_VERSION_NOT_SUPPORTED (0xB027): A handshake handshake message specified an unsupported version.

TLS_HANDSHAKE_STATELESS_REJECT (0xB048): A handshake handshake message resulted in a stateless reject.

TLS_NO_SUPPORT (0xB028): There was no intersection between the crypto primitives supported by the peer and ourselves.

TLS_TOO_MANY_REJECTS (0xB029): The server rejected our client hello messages too many times.

TLS_PROOF_INVALID (0xB02a): The client rejected the server's certificate chain or signature.

TLS_DUPLICATE_TAG (0xB02b): A handshake message was received with a duplicate tag.

TLS_ENCRYPTION_LEVEL_INCORRECT (0xB02c): A handshake message was received with the wrong encryption level (i.e. it should have been encrypted but was not.)

TLS_SERVER_CONFIG_EXPIRED (0xB02d): The server config for a server has expired.

TLS_SYMMETRIC_KEY_SETUP_FAILED (0xB035): We failed to set up the symmetric keys for a connection.

TLS_MESSAGE_WHILE_VALIDATING_CLIENT_HELLO (0xB036): A handshake message arrived, but we are still validating the previous handshake message.

TLS_UPDATE_BEFORE_HANDSHAKE_COMPLETE (0xB041): A server config update arrived before the handshake is complete.

TLS_CLIENT_HELLO_TOO_LARGE (0xB05a): ClientHello cannot fit in one packet.

11. IANA Considerations

This document has no IANA actions. Yet.

12. References

12.1. Normative References

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-18 (work in progress), October 2016.

[QUIC-TRANSPORT]
Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport".

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

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

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

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <http://www.rfc-editor.org/info/rfc7230>.

[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, <http://www.rfc-editor.org/info/rfc7301>.

[RFC7685] Langley, A., "A Transport Layer Security (TLS) ClientHello Padding Extension", RFC 7685, DOI 10.17487/RFC7685, October 2015, <http://www.rfc-editor.org/info/rfc7685>.

12.2. 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>.

[QUIC-HTTP] Bishop, M., Ed., "Hypertext Transfer Protocol (HTTP) over QUIC".

[QUIC-RECOVERY] Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control".

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <http://www.rfc-editor.org/info/rfc793>.

[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, <http://www.rfc-editor.org/info/rfc5280>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <http://www.rfc-editor.org/info/rfc7540>.

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

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.

C.1. 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.2. 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