Workgroup: QUIC
Internet-Draft: draft-ietf-quic-tls-26
Published: 21 February 2020
Intended Status: Standards Track
Expires: 24 August 2020
Authors: M. Thomson, Ed.
Mozilla
S. Turner, Ed.
sn3rd

Using TLS to Secure QUIC

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.

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 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 24 August 2020.

Table of Contents

1. Introduction

This document describes how QUIC [QUIC-TRANSPORT] is secured using 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, using a zero round trip setup.

This document describes how TLS acts as a security component of QUIC.

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, though a newer version could be used (see Section 4.2).

2.1. 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.

Internally, TLS is a layered protocol, with the structure shown in Figure 1.

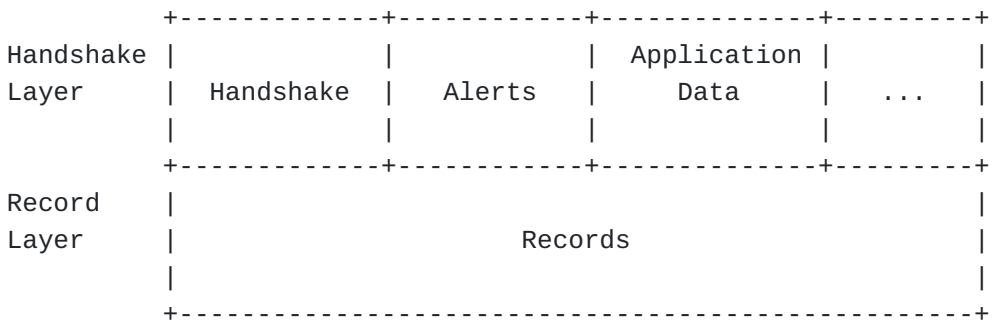

Figure 1: TLS Layers

Each Handshake layer message (e.g., Handshake, Alerts, and Application Data) is carried as a series of typed TLS records by the Record layer. Records are individually cryptographically protected and then transmitted over a reliable transport (typically TCP) which provides sequencing and guaranteed delivery.

The TLS authenticated key exchange occurs between two endpoints: 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 over either finite fields or elliptic curves ((EC)DHE) key exchanges. PSK is the basis for 0-RTT; the latter provides perfect forward secrecy (PFS) when the (EC)DHE 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 resistant to tampering by attackers and it produces shared secrets that cannot be controlled by either participating peer.

TLS 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 handshake with 0-RTT application data is shown in Figure 2. Note that this omits the EndOfEarlyData message, which is not used in QUIC (see Section 8.3). Likewise, neither ChangeCipherSpec nor KeyUpdate messages are used by QUIC; ChangeCipherSpec is redundant in TLS 1.3 and QUIC has defined its own key update mechanism Section 6.

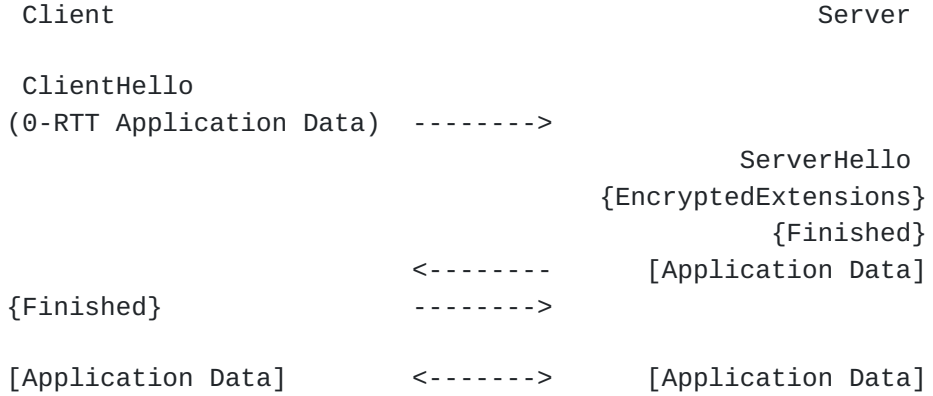

Figure 2: TLS Handshake with 0-RTT

Data is protected using a number of encryption levels:

- Initial Keys
- Early Data (0-RTT) Keys
- Handshake Keys
- Application Data (1-RTT) Keys

Application Data may appear only in the Early Data and Application Data levels. Handshake and Alert messages may appear in any level.

The 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.

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 handshake [TLS13], but instead of carrying TLS records over QUIC (as with TCP), TLS Handshake and Alert messages are carried directly over the QUIC transport, which takes over the responsibilities of the TLS record layer, as shown in Figure 3.

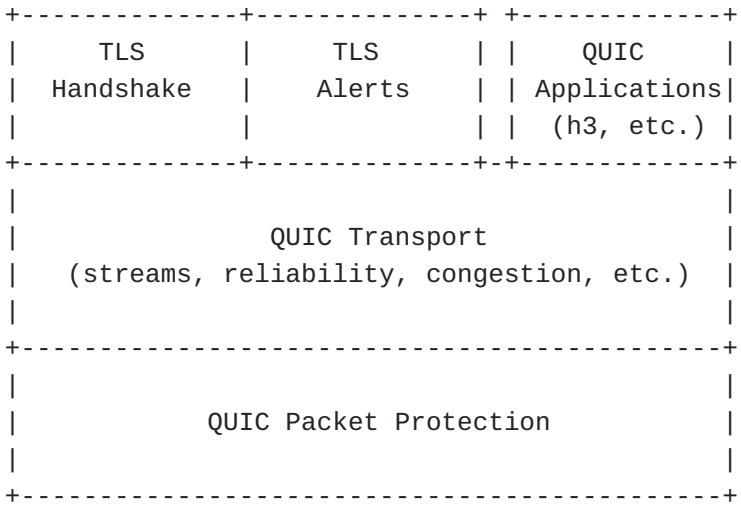

Figure 3: QUIC Layers

QUIC also relies on TLS for authentication and negotiation of parameters that are critical to security and performance.

Rather than a strict layering, these two protocols cooperate: QUIC uses the TLS handshake; TLS uses the reliability, ordered delivery, and record layer provided by QUIC.

At a high level, there are two main interactions between the TLS and QUIC components:

- The TLS component sends and receives messages via the QUIC component, with QUIC providing a reliable stream abstraction to TLS.

- The TLS component provides a series of updates to the QUIC component, including (a) new packet protection keys to install (b) state changes such as handshake completion, the server certificate, etc.

Figure 4 shows these interactions in more detail, with the QUIC packet protection being called out specially.

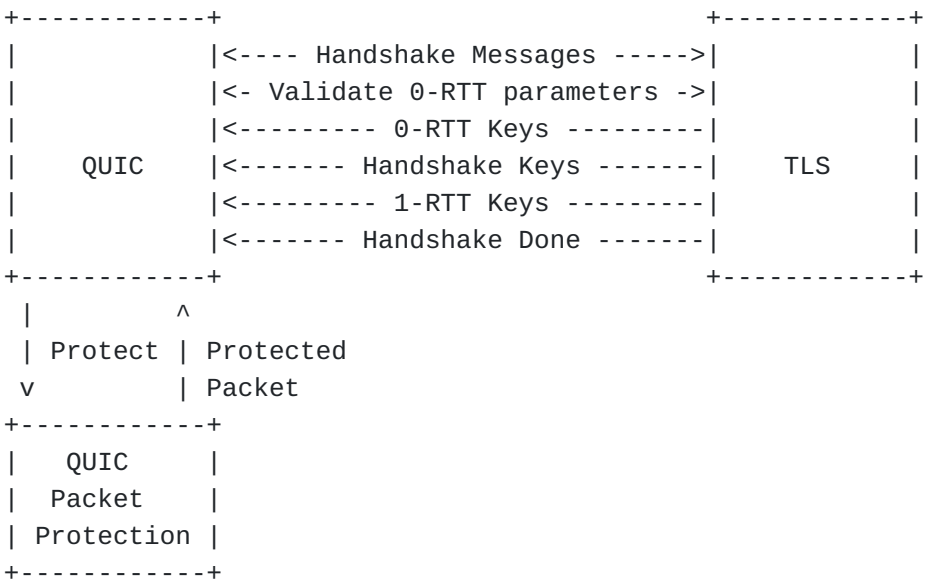

Figure 4: QUIC and TLS Interactions

Unlike TLS over TCP, QUIC applications which want to send data do not send it through TLS "application_data" records. Rather, they send it as QUIC STREAM frames or other frame types which are then carried in QUIC packets.

4. Carrying TLS Messages

QUIC carries TLS handshake data in CRYPTO frames, each of which consists of a contiguous block of handshake data identified by an offset and length. Those frames are packaged into QUIC packets and encrypted under the current TLS encryption level. As with TLS over

TCP, once TLS handshake data has been delivered to QUIC, it is QUIC's responsibility to deliver it reliably. Each chunk of data that is produced by TLS is associated with the set of keys that TLS is currently using. If QUIC needs to retransmit that data, it MUST use the same keys even if TLS has already updated to newer keys.

One important difference between TLS records (used with TCP) and QUIC CRYPTO frames is that in QUIC multiple frames may appear in the same QUIC packet as long as they are associated with the same encryption level. For instance, an implementation might bundle a Handshake message and an ACK for some Handshake data into the same packet.

Some frames are prohibited in different encryption levels, others cannot be sent. The rules here generalize those of TLS, in that frames associated with establishing the connection can usually appear at any encryption level, whereas those associated with transferring data can only appear in the 0-RTT and 1-RTT encryption levels:

- PADDING and PING frames MAY appear in packets of any encryption level.

- CRYPTO frames and CONNECTION_CLOSE frames signaling errors at the QUIC layer (type 0x1c) MAY appear in packets of any encryption level except 0-RTT.

- CONNECTION_CLOSE frames signaling application errors (type 0x1d) MUST only be sent in packets at the 1-RTT encryption level.

- ACK frames MAY appear in packets of any encryption level other than 0-RTT, but can only acknowledge packets which appeared in that packet number space.

- All other frame types MUST only be sent in the 0-RTT and 1-RTT levels.

Note that it is not possible to send the following frames in 0-RTT for various reasons: ACK, CRYPTO, HANDSHAKE_DONE, NEW_TOKEN, PATH_RESPONSE, and RETIRE_CONNECTION_ID.

Because packets could be reordered on the wire, QUIC uses the packet type to indicate which level a given packet was encrypted under, as shown in Table 1. When multiple packets of different encryption levels need to be sent, endpoints SHOULD use coalesced packets to send them in the same UDP datagram.

Packet Type	Encryption Level	PN Space
Initial	Initial secrets	Initial

Packet Type	Encryption Level	PN Space
0-RTT Protected	0-RTT	0/1-RTT
Handshake	Handshake	Handshake
Retry	N/A	N/A
Version Negotiation	N/A	N/A
Short Header	1-RTT	0/1-RTT

Table 1: Encryption Levels by Packet Type

Section 17 of [QUIC-TRANSPORT] shows how packets at the various encryption levels fit into the handshake process.

4.1. Interface to TLS

As shown in Figure 4, the interface from QUIC to TLS consists of four primary functions:

- Sending and receiving handshake messages
- Processing stored transport and application state from a resumed session and determining if it is valid to accept early data
- Rekeying (both transmit and receive)
- Handshake state updates

Additional functions might be needed to configure TLS.

4.1.1. Handshake Complete

In this document, the TLS handshake is considered complete when the TLS stack has reported that the handshake is complete. This happens when the TLS stack has both sent a Finished message and verified the peer's Finished message. Verifying the peer's Finished provides the endpoints with an assurance that previous handshake messages have not been modified. Note that the handshake does not complete at both endpoints simultaneously. Consequently, any requirement that is based on the completion of the handshake depends on the perspective of the endpoint in question.

4.1.2. Handshake Confirmed

In this document, the TLS handshake is considered confirmed at the server when the handshake completes. At the client, the handshake is considered confirmed when a HANDSHAKE_DONE frame is received.

A client MAY consider the handshake to be confirmed when it receives an acknowledgement for a 1-RTT packet. This can be implemented by recording the lowest packet number sent with 1-RTT keys, and comparing it to the Largest Acknowledged field in any received 1-RTT

ACK frame: once the latter is greater than or equal to the former, the handshake is confirmed.

4.1.3. Sending and Receiving Handshake Messages

In order to drive the handshake, TLS depends on being able to send and receive handshake messages. 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 8.2) that it wishes to carry.

A QUIC client starts TLS by requesting TLS handshake bytes from TLS. The client acquires handshake bytes before sending its first packet. A QUIC server starts the process by providing TLS with the client's handshake bytes.

At any time, the TLS stack at an endpoint will have a current sending encryption level and receiving encryption level. Each encryption level is associated with a different flow of bytes, which is reliably transmitted to the peer in CRYPTO frames. When TLS provides handshake bytes to be sent, they are appended to the current flow and any packet that includes the CRYPTO frame is protected using keys from the corresponding encryption level.

QUIC takes the unprotected content of TLS handshake records as the content of CRYPTO frames. TLS record protection is not used by QUIC. QUIC assembles CRYPTO frames into QUIC packets, which are protected using QUIC packet protection.

QUIC is only capable of conveying TLS handshake records in CRYPTO frames. TLS alerts are turned into QUIC CONNECTION_CLOSE error codes; see Section 4.9. TLS application data and other message types cannot be carried by QUIC at any encryption level and is an error if they are received from the TLS stack.

When an endpoint receives a QUIC packet containing a CRYPTO frame from the network, it proceeds as follows:

* If the packet was in the TLS receiving encryption level, sequence the data into the input flow as usual. As with STREAM frames, the offset is used to find the proper location in the data sequence. If the result of this process is that new data is available, then it is delivered to TLS in order.

* If the packet is from a previously installed encryption level, it MUST not contain data which extends past the end of previously received data in that flow. Implementations MUST treat any

violations of this requirement as a connection error of type PROTOCOL_VIOLATION.

* If the packet is from a new encryption level, it is saved for later processing by TLS. Once TLS moves to receiving from this encryption level, saved data can be provided. When providing data from any new encryption level to TLS, if there is data from a previous encryption level that TLS has not consumed, this MUST be treated as a connection error of type PROTOCOL_VIOLATION.

Each time that TLS is provided with new data, new handshake bytes are requested from TLS. TLS might not provide any bytes 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 bytes 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 in CRYPTO streams. In the same way that is done during the handshake, new data is requested from TLS after providing received data.

4.1.4. Encryption Level Changes

As keys for new encryption levels become available, TLS provides QUIC with those keys. Separately, as keys at a given encryption level become available to TLS, TLS indicates to QUIC that reading or writing keys at that encryption level are available. These events are not asynchronous; they always occur immediately after TLS is provided with new handshake bytes, or after TLS produces handshake bytes.

TLS provides QUIC with three items as a new encryption level becomes available:

* A secret

* An Authenticated Encryption with Associated Data (AEAD) function

* A Key Derivation Function (KDF)

These values are based on the values that TLS negotiates and are used by QUIC to generate packet and header protection keys (see Section 5 and Section 5.4).

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 bytes, the TLS stack might signal the change to 0-RTT keys. On the server, after receiving handshake bytes that contain a ClientHello message, a TLS server might signal that 0-RTT keys are available.

Although TLS only uses one encryption level at a time, QUIC may use more than one level. For instance, after sending its Finished message (using a CRYPTO frame at the Handshake encryption level) an endpoint can send STREAM data (in 1-RTT encryption). If the Finished message is lost, the endpoint uses the Handshake encryption level to retransmit the lost message. Reordering or loss of packets can mean that QUIC will need to handle packets at multiple encryption levels. During the handshake, this means potentially handling packets at higher and lower encryption levels than the current encryption level used by TLS.

In particular, server implementations need to be able to read packets at the Handshake encryption level at the same time as the 0-RTT encryption level. A client could interleave ACK frames that are protected with Handshake keys with 0-RTT data and the server needs to process those acknowledgments in order to detect lost Handshake packets.

QUIC also needs access to keys that might not ordinarily be available to a TLS implementation. For instance, a client might need to acknowledge Handshake packets before it is ready to send CRYPTO frames at that encryption level. TLS therefore needs to provide keys to QUIC before it might produce them for its own use.

4.1.5. TLS Interface Summary

Figure 5 summarizes the exchange between QUIC and TLS for both client and server. Each arrow is tagged with the encryption level used for that transmission.

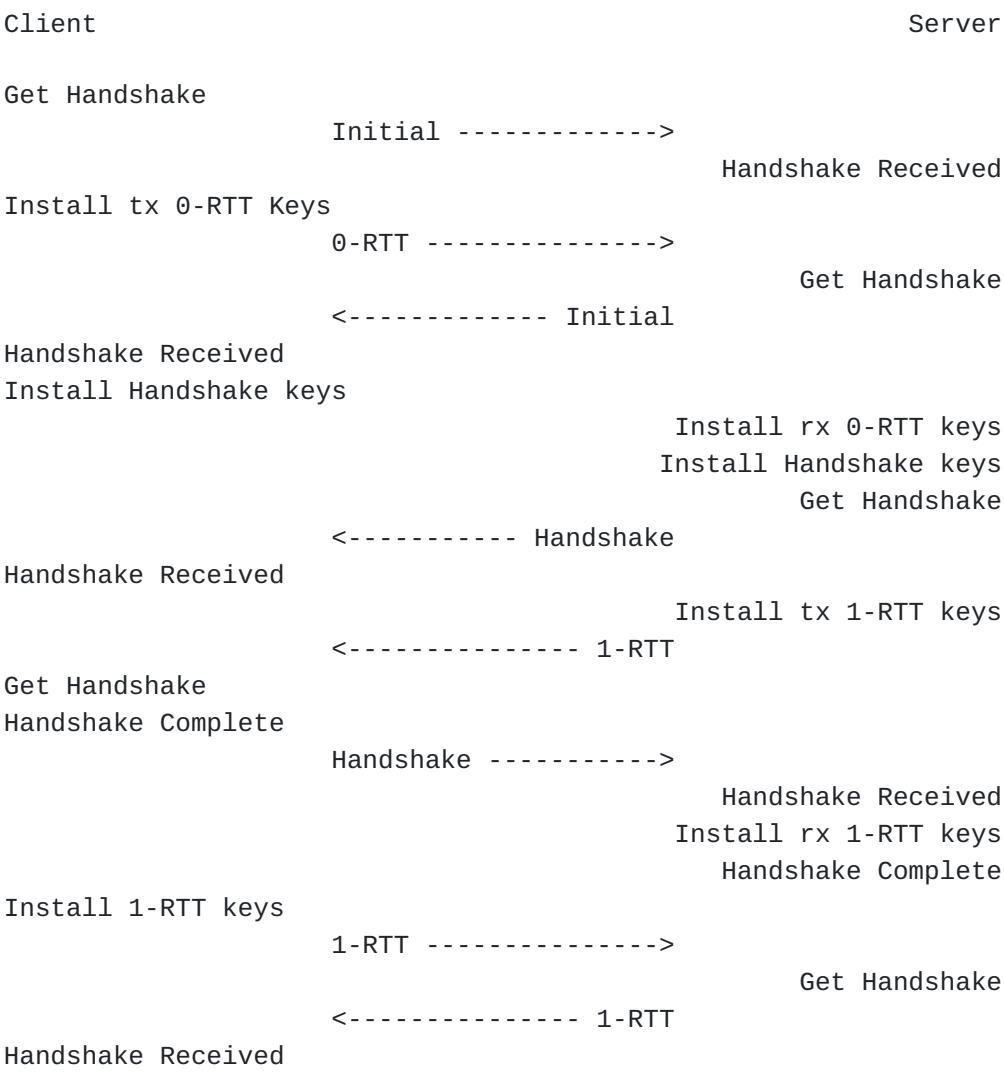

Figure 5: Interaction Summary between QUIC and TLS

Figure 5 shows the multiple packets that form a single "flight" of messages being processed individually, to show what incoming messages trigger different actions. New handshake messages are requested after all incoming packets have been processed. This process might vary depending on how QUIC implementations and the packets they receive are structured.

4.2. 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.

4.3. ClientHello Size

The first Initial packet from a client contains the start or all of its first cryptographic handshake message, which for TLS is the ClientHello. Servers might need to parse the entire ClientHello (e.g., to access extensions such as Server Name Identification (SNI) or Application Layer Protocol Negotiation (ALPN)) in order to decide whether to accept the new incoming QUIC connection. If the ClientHello spans multiple Initial packets, such servers would need to buffer the first received fragments, which could consume excessive resources if the client's address has not yet been validated. To avoid this, servers MAY use the Retry feature (see Section 8.1 of [QUIC-TRANSPORT]) to only buffer partial ClientHello messages from clients with a validated address.

QUIC packet and framing add at least 36 bytes of overhead to the ClientHello message. That overhead increases if the client chooses a connection ID without zero length. Overheads also do not include the token or a connection ID longer than 8 bytes, both of which might be required if a server sends a Retry packet.

A typical TLS ClientHello can easily fit into a 1200 byte packet. However, in addition to the overheads added by QUIC, there are several variables that could cause this limit to be exceeded. Large session tickets, 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, in addition to connection IDs and tokens, the size of TLS session tickets can have an effect on a client's ability to connect efficiently. Minimizing the size of these values increases the probability that clients can use them and still fit their ClientHello message in their first Initial packet.

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.

4.4. 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 (as defined in Section 4.6.2 of [TLS13]), because the multiplexing offered by QUIC prevents clients from correlating the certificate request with the application-level event that triggered it (see [HTTP2-TLS13]). More specifically, servers MUST NOT send post-handshake TLS CertificateRequest messages and clients MUST treat receipt of such messages as a connection error of type PROTOCOL_VIOLATION.

4.5. Enabling 0-RTT

To communicate their willingness to process 0-RTT data, servers send a NewSessionTicket message that contains the "early_data" extension with a max_early_data_size of 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. Servers MUST NOT send the "early_data" extension with a max_early_data_size set to any value other than 0xffffffff. A client MUST treat receipt of a NewSessionTicket that contains an "early_data" extension with any other value as a connection error of type PROTOCOL_VIOLATION.

A client that wishes to send 0-RTT packets uses the "early_data" extension in the ClientHello message of a subsequent handshake (see Section 4.2.10 of [TLS13]). It then sends the application data in 0-RTT packets.

4.6. Accepting and Rejecting 0-RTT

A server accepts 0-RTT by sending an early_data extension in the EncryptedExtensions (see Section 4.2.10 of [TLS13]). The server then processes and acknowledges the 0-RTT packets that it receives.

A server rejects 0-RTT by sending the EncryptedExtensions without an early_data extension. A server will always reject 0-RTT if it sends a TLS HelloRetryRequest. When rejecting 0-RTT, a server MUST NOT process any 0-RTT packets, even if it could. When 0-RTT was rejected, a client SHOULD treat receipt of an acknowledgement for a 0-RTT packet as a connection error of type PROTOCOL_VIOLATION, if it is able to detect the condition.

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.

A client MAY attempt to send 0-RTT again if it receives a Retry or Version Negotiation packet. These packets do not signify rejection of 0-RTT.

4.7. Validating 0-RTT Configuration

When a server receives a ClientHello with the "early_data" extension, it has to decide whether to accept or reject early data from the client. Some of this decision is made by the TLS stack (e.g., checking that the cipher suite being resumed was included in the ClientHello; see Section 4.2.10 of [TLS13]). Even when the TLS stack has no reason to reject early data, the QUIC stack or the application protocol using QUIC might reject early data because the configuration of the transport or application associated with the resumed session is not compatible with the server's current configuration.

QUIC requires additional transport state to be associated with a 0-RTT session ticket. One common way to implement this is using stateless session tickets and storing this state in the session ticket. Application protocols that use QUIC might have similar requirements regarding associating or storing state. This associated state is used for deciding whether early data must be rejected. For example, HTTP/3 ([QUIC-HTTP]) settings determine how early data from the client is interpreted. Other applications using QUIC could have different requirements for determining whether to accept or reject early data.

4.8. HelloRetryRequest

In TLS over TCP, the HelloRetryRequest feature (see Section 4.1.4 of [TLS13]) can be used to correct a client's incorrect KeyShare extension as well as for a stateless round-trip check. From the perspective of QUIC, this just looks like additional messages carried in the Initial encryption level. Although it is in principle possible to use this feature for address verification in QUIC, QUIC implementations SHOULD instead use the Retry feature (see Section 8.1 of [QUIC-TRANSPORT]). HelloRetryRequest is still used to request key shares.

4.9. TLS Errors

If TLS experiences an error, it generates an appropriate alert as defined in Section 6 of [TLS13].

A TLS alert is turned into a QUIC connection error by converting the one-byte alert description into a QUIC error code. The alert description is added to 0x100 to produce a QUIC error code from the range reserved for CRYPTO_ERROR. The resulting value is sent in a QUIC CONNECTION_CLOSE frame.

The alert level of all TLS alerts is "fatal"; a TLS stack MUST NOT generate alerts at the "warning" level.

4.10. Discarding Unused Keys

After QUIC moves to a new encryption level, packet protection keys for previous encryption levels can be discarded. This occurs several times during the handshake, as well as when keys are updated; see Section 6.

Packet protection keys are not discarded immediately when new keys are available. If packets from a lower encryption level contain CRYPTO frames, frames that retransmit that data MUST be sent at the same encryption level. Similarly, an endpoint generates acknowledgements for packets at the same encryption level as the packet being acknowledged. Thus, it is possible that keys for a lower encryption level are needed for a short time after keys for a newer encryption level are available.

An endpoint cannot discard keys for a given encryption level unless it has both received and acknowledged all CRYPTO frames for that encryption level and when all CRYPTO frames for that encryption level have been acknowledged by its peer. However, this does not guarantee that no further packets will need to be received or sent at that encryption level because a peer might not have received all the acknowledgements necessary to reach the same state.

Though an endpoint might retain older keys, new data MUST be sent at the highest currently-available encryption level. Only ACK frames and retransmissions of data in CRYPTO frames are sent at a previous encryption level. These packets MAY also include PADDING frames.

4.10.1. Discarding Initial Keys

Packets protected with Initial secrets (Section 5.2) are not authenticated, meaning that an attacker could spoof packets with the intent to disrupt a connection. To limit these attacks, Initial packet protection keys can be discarded more aggressively than other keys.

The successful use of Handshake packets indicates that no more Initial packets need to be exchanged, as these keys can only be produced after receiving all CRYPTO frames from Initial packets. Thus, a client MUST discard Initial keys when it first sends a

Handshake packet and a server MUST discard Initial keys when it first successfully processes a Handshake packet. Endpoints MUST NOT send Initial packets after this point.

This results in abandoning loss recovery state for the Initial encryption level and ignoring any outstanding Initial packets.

4.10.2. Discarding Handshake Keys

An endpoint MUST discard its handshake keys when the TLS handshake is confirmed (Section 4.1.2). The server MUST send a HANDSHAKE_DONE frame as soon as it completes the handshake.

4.10.3. Discarding 0-RTT Keys

0-RTT and 1-RTT packets share the same packet number space, and clients do not send 0-RTT packets after sending a 1-RTT packet (Section 5.6).

Therefore, a client SHOULD discard 0-RTT keys as soon as it installs 1-RTT keys, since they have no use after that moment.

Additionally, a server MAY discard 0-RTT keys as soon as it receives a 1-RTT packet. However, due to packet reordering, a 0-RTT packet could arrive after a 1-RTT packet. Servers MAY temporarily retain 0-RTT keys to allow decrypting reordered packets without requiring their contents to be retransmitted with 1-RTT keys. After receiving a 1-RTT packet, servers MUST discard 0-RTT keys within a short time; the RECOMMENDED time period is three times the Probe Timeout (PTO, see [QUIC-RECOVERY]). A server MAY discard 0-RTT keys earlier if it determines that it has received all 0-RTT packets, which can be done by keeping track of missing packet numbers.

5. Packet Protection

As with TLS over TCP, QUIC protects packets with keys derived from the TLS handshake, using the AEAD algorithm negotiated by TLS.

5.1. Packet Protection Keys

QUIC derives packet protection keys in the same way that TLS derives record protection keys.

Each encryption level has separate secret values for protection of packets sent in each direction. These traffic secrets are derived by TLS (see Section 7.1 of [TLS13]) and are used by QUIC for all encryption levels except the Initial encryption level. The secrets for the Initial encryption level are computed based on the client's initial Destination Connection ID, as described in Section 5.2.

The keys used for packet protection are computed from the TLS secrets using the KDF provided by TLS. In TLS 1.3, the HKDF-Expand-Label function described in Section 7.1 of [TLS13] is used, using the hash function from the negotiated cipher suite. Other versions of TLS MUST provide a similar function in order to be used with QUIC.

The current encryption level secret and the label "quic key" are input to the KDF to produce the AEAD key; the label "quic iv" is used to derive the IV; see Section 5.3. The header protection key uses the "quic hp" label; see Section 5.4. Using these labels provides key separation between QUIC and TLS; see Section 9.5.

The KDF used for initial secrets is always the HKDF-Expand-Label function from TLS 1.3 (see Section 5.2).

5.2. Initial Secrets

Initial packets are protected with a secret derived from the Destination Connection ID field from the client's Initial packet. Specifically:

```
initial_salt = 0xc3eef712c72ebb5a11a7d2432bb46365bef9f502
initial_secret = HKDF-Extract(initial_salt,
                              client_dst_connection_id)

client_initial_secret = HKDF-Expand-Label(initial_secret,
                                          "client in", "",
                                          Hash.length)
server_initial_secret = HKDF-Expand-Label(initial_secret,
                                          "server in", "",
                                          Hash.length)
```

The hash function for HKDF when deriving initial secrets and keys is SHA-256 [SHA].

The connection ID used with HKDF-Expand-Label is the Destination Connection ID in the Initial packet sent by the client. This will be a randomly-selected value unless the client creates the Initial packet after receiving a Retry packet, where the Destination Connection ID is selected by the server.

The value of initial_salt is a 20 byte 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 packets from future versions.

The HKDF-Expand-Label function defined in TLS 1.3 MUST be used for Initial packets even where the TLS versions offered do not include TLS 1.3.

The secrets used for protecting Initial packets change when a server sends a Retry packet to use the connection ID value selected by the server. The secrets do not change when a client changes the Destination Connection ID it uses in response to an Initial packet from the server.

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 Initial 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.

Appendix A contains test vectors for packet encryption.

5.3. AEAD Usage

The Authentication Encryption with Associated Data (AEAD) [AEAD] function used for QUIC packet protection is the 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.

Packets are protected prior to applying header protection (Section 5.4). The unprotected packet header is part of the associated data (A). When removing packet protection, an endpoint first removes the header protection.

All QUIC packets other than Version Negotiation and Retry 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 Destination Connection ID in the client's first Initial packet (see Section 5.2). This provides protection against off-path attackers and robustness against QUIC version unaware middleboxes, but not against on-path attackers.

QUIC can use any of the ciphersuites defined in [TLS13] with the exception of TLS_AES_128_CCM_8_SHA256. A ciphersuite MUST NOT be negotiated unless a header protection scheme is defined for the ciphersuite. This document defines a header protection scheme for all ciphersuites defined in [TLS13] aside from TLS_AES_128_CCM_8_SHA256. These ciphersuites have a 16-byte authentication tag and produce an output 16 bytes larger than their input.

Note:

> An endpoint MUST NOT reject a ClientHello that offers a ciphersuite that it does not support, or it would be impossible to deploy a new ciphersuite. This also applies to TLS_AES_128_CCM_8_SHA256.

The key and IV for the packet are computed as described in Section 5.1. The nonce, N, is formed by combining the packet protection IV with the packet number. The 62 bits of the reconstructed QUIC packet number in network byte order are 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 byte in either the short or long header, up to and including the unprotected packet number.

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

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

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) prior to exceeding any limit set for the AEAD that is in use.

5.4. Header Protection

Parts of QUIC packet headers, in particular the Packet Number field, are protected using a key that is derived separate to the packet protection key and IV. The key derived using the "quic hp" label is used to provide confidentiality protection for those fields that are not exposed to on-path elements.

This protection applies to the least-significant bits of the first byte, plus the Packet Number field. The four least-significant bits of the first byte are protected for packets with long headers; the five least significant bits of the first byte are protected for packets with short headers. For both header forms, this covers the reserved bits and the Packet Number Length field; the Key Phase bit is also protected for packets with a short header.

The same header protection key is used for the duration of the connection, with the value not changing after a key update (see Section 6). This allows header protection to be used to protect the key phase.

This process does not apply to Retry or Version Negotiation packets, which do not contain a protected payload or any of the fields that are protected by this process.

5.4.1. Header Protection Application

Header protection is applied after packet protection is applied (see Section 5.3). The ciphertext of the packet is sampled and used as input to an encryption algorithm. The algorithm used depends on the negotiated AEAD.

The output of this algorithm is a 5 byte mask which is applied to the protected header fields using exclusive OR. The least significant bits of the first byte of the packet are masked by the least significant bits of the first mask byte, and the packet number is masked with the remaining bytes. Any unused bytes of mask that might result from a shorter packet number encoding are unused.

Figure 6 shows a sample algorithm for applying header protection. Removing header protection only differs in the order in which the packet number length (pn_length) is determined.

```
mask = header_protection(hp_key, sample)

pn_length = (packet[0] & 0x03) + 1
if (packet[0] & 0x80) == 0x80:
   # Long header: 4 bits masked
   packet[0] ^= mask[0] & 0x0f
else:
   # Short header: 5 bits masked
   packet[0] ^= mask[0] & 0x1f

# pn_offset is the start of the Packet Number field.
packet[pn_offset:pn_offset+pn_length] ^= mask[1:1+pn_length]
```

Figure 6: Header Protection Pseudocode

Figure 7 shows the protected fields of long and short headers marked with an E. Figure 7 also shows the sampled fields.

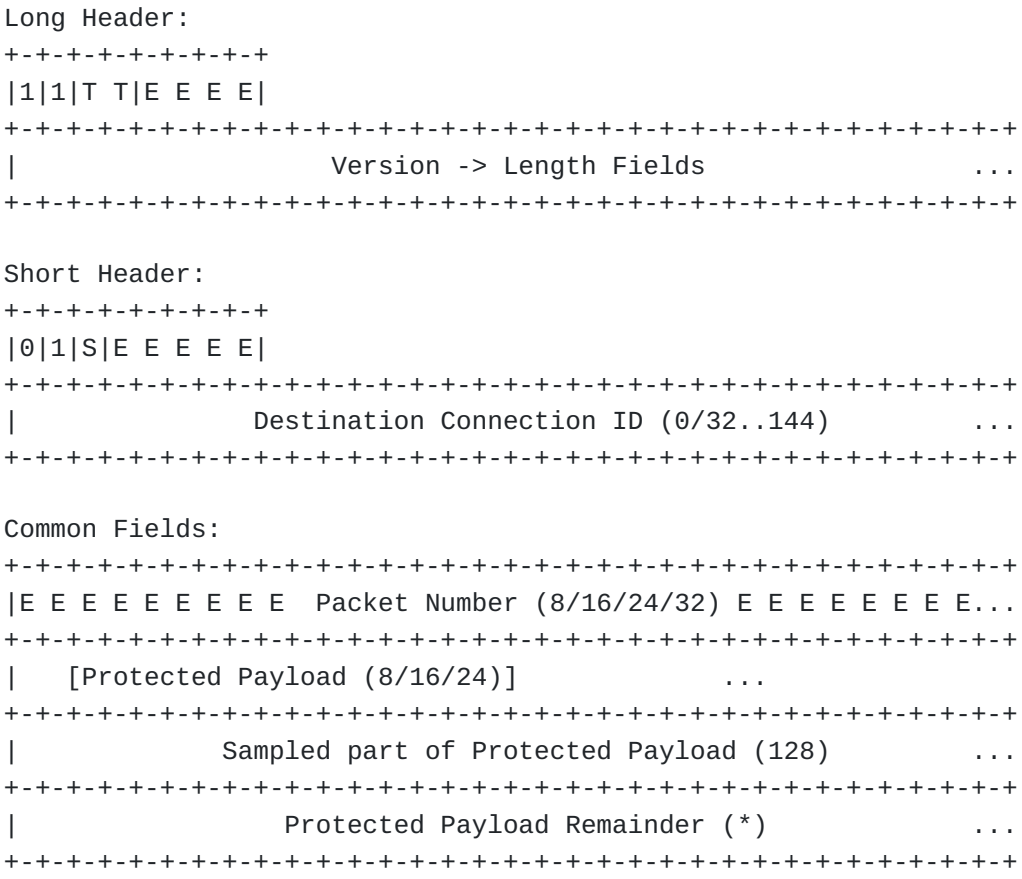

Figure 7: Header Protection and Ciphertext Sample

Before a TLS ciphersuite can be used with QUIC, a header protection algorithm MUST be specified for the AEAD used with that ciphersuite. This document defines algorithms for AEAD_AES_128_GCM, AEAD_AES_128_CCM, AEAD_AES_256_GCM (all AES AEADs are defined in [AEAD]), and AEAD_CHACHA20_POLY1305 [CHACHA]. Prior to TLS selecting a ciphersuite, AES header protection is used (Section 5.4.3), matching the AEAD_AES_128_GCM packet protection.

5.4.2. Header Protection Sample

The header protection algorithm uses both the header protection key and a sample of the ciphertext from the packet Payload field.

The same number of bytes are always sampled, but an allowance needs to be made for the endpoint removing protection, which will not know the length of the Packet Number field. In sampling the packet ciphertext, the Packet Number field is assumed to be 4 bytes long (its maximum possible encoded length).

An endpoint MUST discard packets that are not long enough to contain a complete sample.

To ensure that sufficient data is available for sampling, packets are padded so that the combined lengths of the encoded packet number and protected payload is at least 4 bytes longer than the sample required for header protection. The ciphersuites defined in [TLS13] - other than TLS_AES_128_CCM_8_SHA256, for which a header protection scheme is not defined in this document - have 16-byte expansions and 16-byte header protection samples. This results in needing at least 3 bytes of frames in the unprotected payload if the packet number is encoded on a single byte, or 2 bytes of frames for a 2-byte packet number encoding.

The sampled ciphertext for a packet with a short header can be determined by the following pseudocode:

```
sample_offset = 1 + len(connection_id) + 4

sample = packet[sample_offset..sample_offset+sample_length]
```

For example, for a packet with a short header, an 8 byte connection ID, and protected with AEAD_AES_128_GCM, the sample takes bytes 13 to 28 inclusive (using zero-based indexing).

A packet with a long header is sampled in the same way, noting that multiple QUIC packets might be included in the same UDP datagram and that each one is handled separately.

```
sample_offset = 7 + len(destination_connection_id) +
                    len(source_connection_id) +
                    len(payload_length) + 4
if packet_type == Initial:
    sample_offset += len(token_length) +
                     len(token)

sample = packet[sample_offset..sample_offset+sample_length]
```

5.4.3. AES-Based Header Protection

This section defines the packet protection algorithm for AEAD_AES_128_GCM, AEAD_AES_128_CCM, and AEAD_AES_256_GCM. AEAD_AES_128_GCM and AEAD_AES_128_CCM use 128-bit AES [AES] in electronic code-book (ECB) mode. AEAD_AES_256_GCM uses 256-bit AES in ECB mode.

This algorithm samples 16 bytes from the packet ciphertext. This value is used as the input to AES-ECB. In pseudocode:

```
mask = AES-ECB(hp_key, sample)
```

5.4.4. ChaCha20-Based Header Protection

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

The first 4 bytes of the sampled ciphertext are the block counter. A ChaCha20 implementation could take a 32-bit integer in place of a byte sequence, in which case the byte sequence is interpreted as a little-endian value.

The remaining 12 bytes are used as the nonce. A ChaCha20 implementation might take an array of three 32-bit integers in place of a byte sequence, in which case the nonce bytes are interpreted as a sequence of 32-bit little-endian integers.

The encryption mask is produced by invoking ChaCha20 to protect 5 zero bytes. In pseudocode:

```
counter = sample[0..3]
nonce = sample[4..15]
mask = ChaCha20(hp_key, counter, nonce, {0,0,0,0,0})
```

5.5. Receiving Protected Packets

Once an endpoint successfully receives a packet with a given packet number, it MUST discard all packets in the same packet number space 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). 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.

5.6. Use of 0-RTT Keys

If 0-RTT keys are available (see Section 4.5), 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, except that it MUST NOT send ACKs with 0-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; it uses 1-RTT keys to protect acknowledgements of 0-RTT packets. A client MUST NOT attempt to decrypt 0-RTT packets it receives and instead MUST discard them.

Once a client has installed 1-RTT keys, it MUST NOT send any more 0-RTT packets.

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.

5.7. 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. Endpoints in either role MUST NOT decrypt 1-RTT packets from their peer prior to completing the handshake.

Even though 1-RTT keys are available to a server after receiving the first handshake messages from a client, it is missing assurances on the client state:

- The client is not authenticated, unless the server has chosen to use a pre-shared key and validated the client's pre-shared key binder; see Section 4.2.11 of [TLS13].

- The client has not demonstrated liveness, unless a RETRY packet was used.

- Any received 0-RTT data that the server responds to might be due to a replay attack.

Therefore, the server's use of 1-RTT keys MUST be limited to sending data before the handshake is complete. A server MUST NOT process incoming 1-RTT protected packets before the TLS handshake is complete. Because sending acknowledgments indicates that all frames in a packet have been processed, a server cannot send acknowledgments for 1-RTT packets until the TLS handshake is

complete. Received packets protected with 1-RTT keys MAY be stored and later decrypted and used once the handshake is complete.

Note: TLS implementations might provide all 1-RTT secrets prior to handshake completion. Even where QUIC implementations have 1-RTT read keys, those keys cannot be used prior to completing the handshake.

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 its 1-RTT packets coalesced with a handshake packet containing a copy of the CRYPTO frame that carries the Finished message, until one of the handshake packets is acknowledged. This enables immediate server processing for those packets.

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.

5.8. Retry Packet Integrity

Retry packets (see the Retry Packet section of [QUIC-TRANSPORT]) carry a Retry Integrity Tag that provides two properties: it allows discarding packets that have accidentally been corrupted by the network, and it diminishes off-path attackers' ability to send valid Retry packets.

The Retry Integrity Tag is a 128-bit field that is computed as the output of AEAD_AES_128_GCM [AEAD] used with the following inputs:

- The secret key, K, is 128 bits equal to 0x4d32ecdb2a2133c841e4043df27d4430.
- The nonce, N, is 96 bits equal to 0x4d1611d05513a552c587d575.
- The plaintext, P, is empty.
- The associated data, A, is the contents of the Retry Pseudo-Packet, as illustrated in Figure 8:

The secret key and the nonce are values derived by calling HKDF-Expand-Label using 0x656e61e336ae9417f7f0edd8d78d461e2aa7084aba7a14c1e9f726d55709169a as the secret, with labels being "quic key" and "quic iv" (Section 5.1).

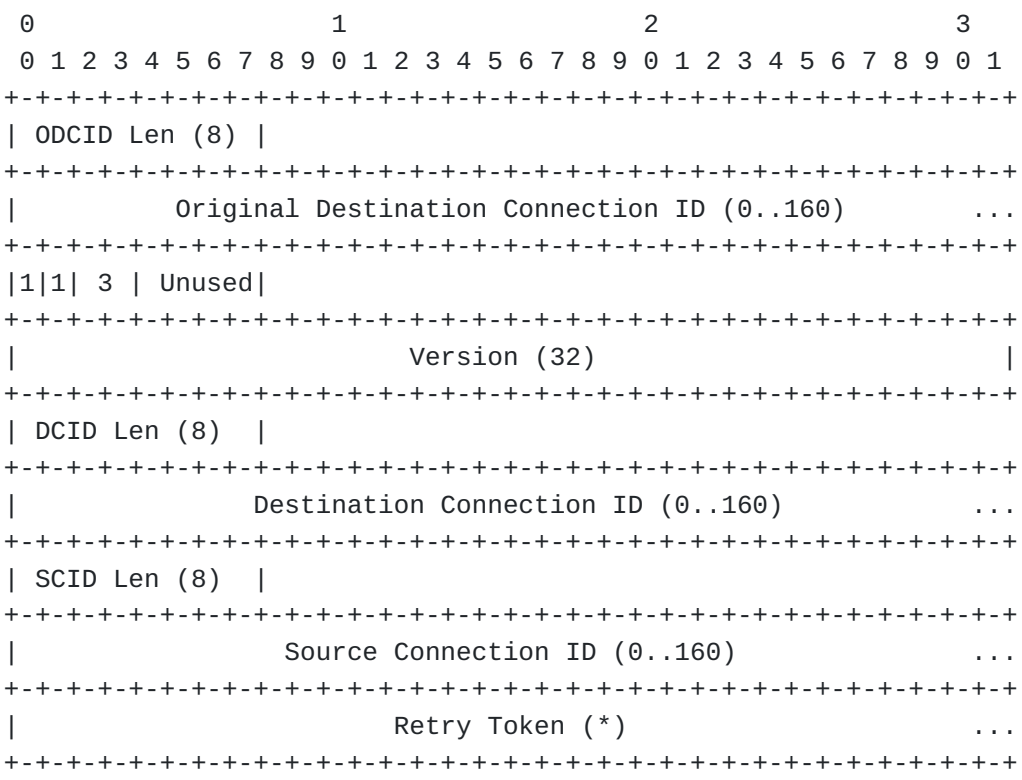

Figure 8: Retry Pseudo-Packet

The Retry Pseudo-Packet is not sent over the wire. It is computed by taking the transmitted Retry packet, removing the Retry Integrity Tag and prepending the two following fields:

ODCID Len: The ODCID Len contains the length in bytes of the Original Destination Connection ID field that follows it, encoded as an 8-bit unsigned integer.

Original Destination Connection ID: The Original Destination Connection ID contains the value of the Destination Connection ID from the Initial packet that this Retry is in response to. The length of this field is given in ODCID Len. The presence of this field mitigates an off-path attacker's ability to inject a Retry packet.

6. Key Update

Once the handshake is confirmed (see Section 4.1.2), an endpoint MAY initiate a key update.

The Key Phase bit indicates which packet protection keys are used to protect the packet. The Key Phase bit is initially set to 0 for the first set of 1-RTT packets and toggled to signal each subsequent key update.

The Key Phase bit allows a recipient to detect a change in keying material without needing to receive the first packet that triggered the change. An endpoint that notices a changed Key Phase bit updates keys and decrypts the packet that contains the changed value.

This mechanism replaces the TLS KeyUpdate message. Endpoints MUST NOT send a TLS KeyUpdate message. Endpoints MUST treat the receipt of a TLS KeyUpdate message as a connection error of type 0x10a, equivalent to a fatal TLS alert of unexpected_message (see Section 4.9).

Figure 9 shows a key update process, where the initial set of keys used (identified with @M) are replaced by updated keys (identified with @N). The value of the Key Phase bit is indicated in brackets [].

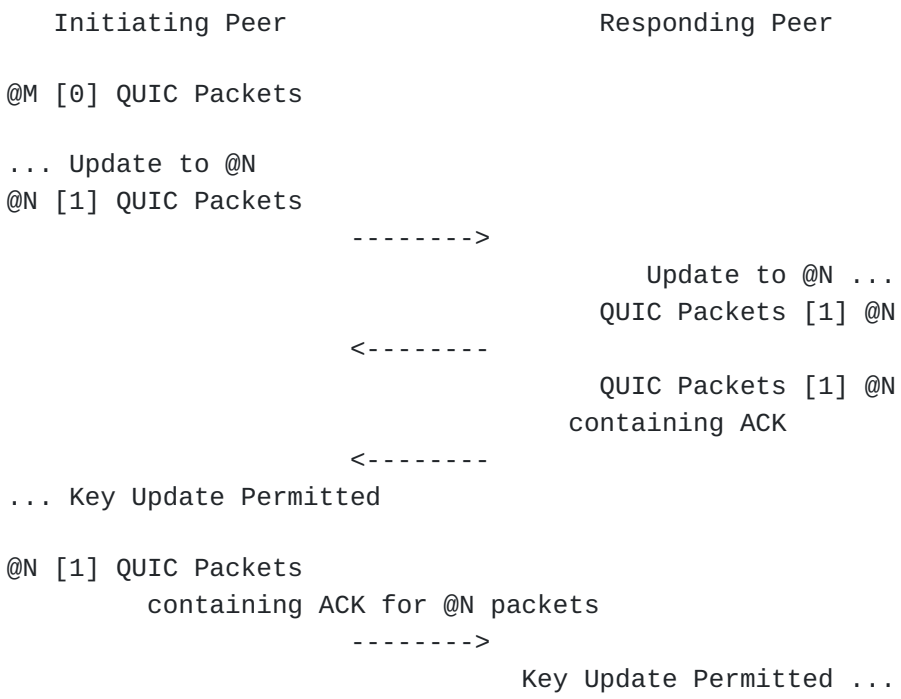

Figure 9: Key Update

6.1. Initiating a Key Update

Endpoints maintain separate read and write secrets for packet protection. An endpoint initiates a key update by updating its packet protection write secret and using that to protect new packets. The endpoint creates a new write secret from the existing write secret as performed in Section 7.2 of [TLS13]. This uses the KDF function provided by TLS with a label of "quic ku". The corresponding key and IV are created from that secret as defined in Section 5.1. The header protection key is not updated.

For example, to update write keys with TLS 1.3, HKDF-Expand-Label is used as:

```
secret_<n+1> = HKDF-Expand-Label(secret_<n>, "quic ku",
                                 "", Hash.length)
```

The endpoint toggles the value of the Key Phase bit and uses the updated key and IV to protect all subsequent packets.

An endpoint MUST NOT initiate a key update prior to having confirmed the handshake (Section 4.1.2). An endpoint MUST NOT initiate a subsequent key update prior unless it has received an acknowledgment for a packet that was sent protected with keys from the current key phase. This ensures that keys are available to both peers before another key update can be initiated. This can be implemented by tracking the lowest packet number sent with each key phase, and the highest acknowledged packet number in the 1-RTT space: once the latter is higher than or equal to the former, another key update can be initiated.

Note: Keys of packets other than the 1-RTT packets are never updated; their keys are derived solely from the TLS handshake state.

The endpoint that initiates a key update also updates the keys that it uses for receiving packets. These keys will be needed to process packets the peer sends after updating.

An endpoint SHOULD retain old keys so that packets sent by its peer prior to receiving the key update can be processed. Discarding old keys too early can cause delayed packets to be discarded. Discarding packets will be interpreted as packet loss by the peer and could adversely affect performance.

6.2. Responding to a Key Update

A peer is permitted to initiate a key update after receiving an acknowledgement of a packet in the current key phase. An endpoint detects a key update when processing a packet with a key phase that differs from the value last used to protect the last packet it sent. To process this packet, the endpoint uses the next packet protection key and IV. See Section 6.3 for considerations about generating these keys.

If a packet is successfully processed using the next key and IV, then the peer has initiated a key update. The endpoint MUST update its send keys to the corresponding key phase in response, as described in Section 6.1. Sending keys MUST be updated before sending an acknowledgement for the packet that was received with updated keys. By acknowledging the packet that triggered the key

update in a packet protected with the updated keys, the endpoint signals that the key update is complete.

An endpoint can defer sending the packet or acknowledgement according to its normal packet sending behaviour; it is not necessary to immediately generate a packet in response to a key update. The next packet sent by the endpoint will use the updated keys. The next packet that contains an acknowledgement will cause the key update to be completed. If an endpoint detects a second update before it has sent any packets with updated keys containing an acknowledgement for the packet that initiated the key update, it indicates that its peer has updated keys twice without awaiting confirmation. An endpoint MAY treat consecutive key updates as a connection error of type KEY_UPDATE_ERROR.

An endpoint that receives an acknowledgement that is carried in a packet protected with old keys where any acknowledged packet was protected with newer keys MAY treat that as a connection error of type KEY_UPDATE_ERROR. This indicates that a peer has received and acknowledged a packet that initiates a key update, but has not updated keys in response.

6.3. Timing of Receive Key Generation

Endpoints responding to an apparent key update MUST NOT generate a timing side-channel signal that might indicate that the Key Phase bit was invalid (see Section 9.3). Endpoints can use dummy packet protection keys in place of discarded keys when key updates are not yet permitted. Using dummy keys will generate no variation in the timing signal produced by attempting to remove packet protection, and results in all packets with an invalid Key Phase bit being rejected.

The process of creating new packet protection keys for receiving packets could reveal that a key update has occurred. An endpoint MAY perform this process as part of packet processing, but this creates a timing signal that can be used by an attacker to learn when key updates happen and thus the value of the Key Phase bit in certain packets. Endpoints MAY instead defer the creation of the next set of receive packet protection keys until some time after a key update completes, up to three times the PTO; see Section 6.5.

Once generated, the next set of packet protection keys SHOULD be retained, even if the packet that was received was subsequently discarded. Packets containing apparent key updates are easy to forge and - while the process of key update does not require significant effort - triggering this process could be used by an attacker for DoS.

For this reason, endpoints MUST be able to retain two sets of packet protection keys for receiving packets: the current and the next. Retaining the previous keys in addition to these might improve performance, but this is not essential.

6.4. Sending with Updated Keys

An endpoint always sends packets that are protected with the newest keys. Keys used for packet protection can be discarded immediately after switching to newer keys.

Packets with higher packet numbers MUST be protected with either the same or newer packet protection keys than packets with lower packet numbers. An endpoint that successfully removes protection with old keys when newer keys were used for packets with lower packet numbers MUST treat this as a connection error of type KEY_UPDATE_ERROR.

6.5. Receiving with Different Keys

For receiving packets during a key update, packets protected with older keys might arrive if they were delayed by the network. Retaining old packet protection keys allows these packets to be successfully processed.

As packets protected with keys from the next key phase use the same Key Phase value as those protected with keys from the previous key phase, it can be necessary to distinguish between the two. This can be done using packet numbers. A recovered packet number that is lower than any packet number from the current key phase uses the previous packet protection keys; a recovered packet number that is higher than any packet number from the current key phase requires the use of the next packet protection keys.

Some care is necessary to ensure that any process for selecting between previous, current, and next packet protection keys does not expose a timing side channel that might reveal which keys were used to remove packet protection. See Section 9.4 for more information.

Alternatively, endpoints can retain only two sets of packet protection keys, swapping previous for next after enough time has passed to allow for reordering in the network. In this case, the Key Phase bit alone can be used to select keys.

An endpoint MAY allow a period of approximately the Probe Timeout (PTO; see [QUIC-RECOVERY]) after a key update before it creates the next set of packet protection keys. These updated keys MAY replace the previous keys at that time. With the caveat that PTO is a subjective measure - that is, a peer could have a different view of the RTT - this time is expected to be long enough that any reordered

packets would be declared lost by a peer even if they were acknowledged and short enough to allow for subsequent key updates.

Endpoints need to allow for the possibility that a peer might not be able to decrypt packets that initiate a key update during the period when it retains old keys. Endpoints SHOULD wait three times the PTO before initiating a key update after receiving an acknowledgment that confirms that the previous key update was received. Failing to allow sufficient time could lead to packets being discarded.

An endpoint SHOULD retain old read keys for no more than three times the PTO. After this period, old read keys and their corresponding secrets SHOULD be discarded.

6.6. Key Update Frequency

Key updates MUST be initiated before usage limits on packet protection keys are exceeded. For the cipher suites mentioned in this document, the limits in Section 5.5 of [TLS13] apply. Other cipher suites MUST define usage limits in order to be used with QUIC.

6.7. Key Update Error Code

The KEY_UPDATE_ERROR error code (0xE) is used to signal errors related to key updates.

7. Security of Initial Messages

Initial packets are not protected with a secret key, so they are subject to potential tampering by an attacker. QUIC provides protection against attackers that cannot read packets, but does not attempt to provide additional protection against attacks where the attacker can observe and inject packets. Some forms of tampering - such as modifying the TLS messages themselves - are detectable, but some - such as modifying ACKs - are not.

For example, an attacker could inject a packet containing an ACK frame that makes it appear that a packet had not been received or to create a false impression of the state of the connection (e.g., by modifying the ACK Delay). Note that such a packet could cause a legitimate packet to be dropped as a duplicate. Implementations SHOULD use caution in relying on any data which is contained in Initial packets that is not otherwise authenticated.

It is also possible for the attacker to tamper with data that is carried in Handshake packets, but because that tampering requires modifying TLS handshake messages, that tampering will cause the TLS handshake to fail.

8. QUIC-Specific Additions to the TLS Handshake

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

8.1. Protocol Negotiation

QUIC requires that the cryptographic handshake provide authenticated protocol negotiation. TLS uses Application Layer Protocol Negotiation (ALPN) [ALPN] to select an application protocol. Unless another mechanism is used for agreeing on an application protocol, endpoints MUST use ALPN for this purpose. When using ALPN, endpoints MUST immediately close a connection (see Section 10.3 in [QUIC-TRANSPORT]) if an application protocol is not negotiated with a no_application_protocol TLS alert (QUIC error code 0x178, see Section 4.9). While [ALPN] only specifies that servers use this alert, QUIC clients MUST also use it to terminate a connection when ALPN negotiation fails.

An application 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. The server MUST treat the inability to select a compatible application protocol as a connection error of type 0x178 (no_application_protocol). Similarly, a client MUST treat the selection of an incompatible application protocol by a server as a connection error of type 0x178.

8.2. QUIC Transport Parameters Extension

QUIC transport parameters are carried in a TLS extension. Different versions of QUIC might define a different method for negotiating transport configuration.

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

```
enum {
   quic_transport_parameters(0xffa5), (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 is carried in the ClientHello and the EncryptedExtensions messages during the handshake. Endpoints MUST send the quic_transport_parameters extension; endpoints that receive ClientHello or EncryptedExtensions

messages without the quic_transport_parameters extension MUST close the connection with an error of type 0x16d (equivalent to a fatal TLS missing_extension alert, see Section 4.9).

While the transport parameters are technically available prior to the completion of the handshake, they cannot be fully trusted until the handshake completes, and reliance on them should be minimized. However, any tampering with the parameters will cause the handshake to fail.

Endpoints MUST NOT send this extension in a TLS connection that does not use QUIC (such as the use of TLS with TCP defined in [TLS13]). A fatal unsupported_extension alert MUST be sent by an implementation that supports this extension if the extension is received when the transport is not QUIC.

8.3. Removing the EndOfEarlyData Message

The TLS EndOfEarlyData message is not used with QUIC. QUIC does not rely on this message to mark the end of 0-RTT data or to signal the change to Handshake keys.

Clients MUST NOT send the EndOfEarlyData message. A server MUST treat receipt of a CRYPTO frame in a 0-RTT packet as a connection error of type PROTOCOL_VIOLATION.

As a result, EndOfEarlyData does not appear in the TLS handshake transcript.

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. Replay Attacks with 0-RTT

As described in Section 8 of [TLS13], use of TLS early data comes with an exposure to replay attack. The use of 0-RTT in QUIC is similarly vulnerable to replay attack.

Endpoints MUST implement and use the replay protections described in [TLS13], however it is recognized that these protections are imperfect. Therefore, additional consideration of the risk of replay is needed.

QUIC is not vulnerable to replay attack, except via the application protocol information it might carry. The management of QUIC protocol state based on the frame types defined in [QUIC-TRANSPORT] is not vulnerable to replay. Processing of QUIC frames is idempotent and cannot result in invalid connection states if frames are replayed, reordered or lost. QUIC connections do not produce effects that last beyond the lifetime of the connection, except for those produced by the application protocol that QUIC serves.

Note: TLS session tickets and address validation tokens are used to carry QUIC configuration information between connections. These MUST NOT be used to carry application semantics. The potential for reuse of these tokens means that they require stronger protections against replay.

A server that accepts 0-RTT on a connection incurs a higher cost than accepting a connection without 0-RTT. This includes higher processing and computation costs. Servers need to consider the probability of replay and all associated costs when accepting 0-RTT.

Ultimately, the responsibility for managing the risks of replay attacks with 0-RTT lies with an application protocol. An application protocol that uses QUIC MUST describe how the protocol uses 0-RTT and the measures that are employed to protect against replay attack. An analysis of replay risk needs to consider all QUIC protocol features that carry application semantics.

Disabling 0-RTT entirely is the most effective defense against replay attack.

QUIC extensions MUST describe how replay attacks affect their operation, or prohibit their use in 0-RTT. Application protocols MUST either prohibit the use of extensions that carry application semantics in 0-RTT or provide replay mitigation strategies.

9.2. 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.

QUIC includes three defenses against this attack. First, the packet containing a ClientHello MUST be padded to a minimum size. Second, if responding to an unverified source address, the server is forbidden to send more than three UDP datagrams in its first flight (see Section 8.1 of [QUIC-TRANSPORT]). Finally, because acknowledgements of Handshake packets are authenticated, a blind attacker cannot forge them. Put together, these defenses limit the level of amplification.

9.3. Header Protection Analysis

[NAN] analyzes authenticated encryption algorithms which provide nonce privacy, referred to as "Hide Nonce" (HN) transforms. The general header protection construction in this document is one of those algorithms (HN1). Header protection uses the output of the packet protection AEAD to derive sample, and then encrypts the header field using a pseudorandom function (PRF) as follows:

```
protected_field = field XOR PRF(hp_key, sample)
```

The header protection variants in this document use a pseudorandom permutation (PRP) in place of a generic PRF. However, since all PRPs are also PRFs [IMC], these variants do not deviate from the HN1 construction.

As hp_key is distinct from the packet protection key, it follows that header protection achieves AE2 security as defined in [NAN] and therefore guarantees privacy of field, the protected packet header. Future header protection variants based on this construction MUST use a PRF to ensure equivalent security guarantees.

Use of the same key and ciphertext sample more than once risks compromising header protection. Protecting two different headers with the same key and ciphertext sample reveals the exclusive OR of the protected fields. 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 headers, the header is transitively authenticated using packet protection; the entire packet header is part of the authenticated additional data. Protected fields that are falsified or modified can only be detected once the packet protection is removed.

9.4. Header Protection Timing Side-Channels

An attacker could guess values for packet numbers or Key Phase and have an endpoint confirm guesses through timing side channels. Similarly, guesses for the packet number length can be trialed and exposed. 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 header 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.

During a key update, the time taken to generate new keys could reveal through timing side-channels that a key update has occurred. Alternatively, where an attacker injects packets this side-channel could reveal the value of the Key Phase on injected packets. After receiving a key update, an endpoint SHOULD generate and save the next set of receive packet protection keys, as described in Section 6.3. By generating new keys before a key update is received, receipt of packets will not create timing signals that leak the value of the Key Phase.

This depends on not doing this key generation during packet processing and it can require that endpoints maintain three sets of packet protection keys for receiving: for the previous key phase, for the current key phase, and for the next key phase. Endpoints can instead choose to defer generation of the next receive packet protection keys until they discard old keys so that only two sets of receive keys need to be retained at any point in time.

9.5. Key Diversity

In using TLS, the central key schedule of TLS is used. As a result of the TLS handshake messages being integrated into the calculation of secrets, the inclusion of the QUIC transport parameters extension ensures that handshake and 1-RTT keys are not the same as those that might be produced by a server running TLS over TCP. To avoid the possibility of cross-protocol key synchronization, additional measures are provided to improve key separation.

The QUIC packet protection keys and IVs are derived using a different label than the equivalent keys in TLS.

To preserve this separation, a new version of QUIC SHOULD define new labels for key derivation for packet protection key and IV, plus the header protection keys. This version of QUIC uses the string "quic". Other versions can use a version-specific label in place of that string.

The initial secrets use a key that is specific to the negotiated QUIC version. New QUIC versions SHOULD define a new salt value used in calculating initial secrets.

10. IANA Considerations

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

- TLS ExtensionType Values Registry [TLS-REGISTRIES] - IANA is to register the quic_transport_parameters extension found in Section 8.2. The Recommended column is to be marked Yes. The TLS 1.3 Column is to include CH and EE.

- QUIC Transport Error Codes Registry [QUIC-TRANSPORT] - IANA is to register the KEY_UPDATE_ERROR (0xE), as described in Section 6.7.

11. References

11.1. 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)", DOI 10.6028/nist.fips.197, National Institute of Standards and Technology report, November 2001, <https://doi.org/10.6028/nist.fips.197>.

[ALPN] 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>.

[CHACHA] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 8439, DOI 10.17487/RFC8439, June 2018, <https://www.rfc-editor.org/info/rfc8439>.

[QUIC-RECOVERY] Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control", Work in Progress, Internet-Draft, draft-ietf-quic-recovery-26, 21 February 2020, <https://tools.ietf.org/html/draft-ietf-quic-recovery-26>.

[QUIC-TRANSPORT] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-26,

21 February 2020, <https://tools.ietf.org/html/draft-ietf-quic-transport-26>.

[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>.

[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", DOI 10.6028/nist.fips.180-4, National Institute of Standards and Technology report, July 2015, <https://doi.org/10.6028/nist.fips.180-4>.

[TLS-REGISTRIES] Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", RFC 8447, DOI 10.17487/RFC8447, August 2018, <https://www.rfc-editor.org/info/rfc8447>.

[TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

11.2. Informative References

[AEBounds] Luykx, A. and K. Paterson, "Limits on Authenticated Encryption Use in TLS", 8 March 2016, <http://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf>.

[HTTP2-TLS13] Benjamin, D., "Using TLS 1.3 with HTTP/2", Work in Progress, Internet-Draft, draft-ietf-httpbis-http2-tls13-03, 17 October 2019, <http://www.ietf.org/internet-drafts/draft-ietf-httpbis-http2-tls13-03.txt>.

[IMC] Katz, J. and Y. Lindell, "Introduction to Modern Cryptography, Second Edition", ISBN 978-1466570269, 6 November 2014.

[NAN] Bellare, M., Ng, R., and B. Tackmann, "Nonces Are Noticed: AEAD Revisited", DOI 10.1007/978-3-030-26948-7_9, Advances in Cryptology - CRYPTO 2019 pp. 235-265, 2019, <https://doi.org/10.1007/978-3-030-26948-7_9>.

[QUIC-HTTP] Bishop, M., Ed., "Hypertext Transfer Protocol Version 3 (HTTP/3)", Work in Progress, Internet-Draft, draft-ietf-quic-http-26, 21 February 2020, <https://tools.ietf.org/html/draft-ietf-quic-http-26>.

[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>.

Appendix A. Sample Packet Protection

This section shows examples of packet protection so that implementations can be verified incrementally. Samples of Initial packets from both client and server, plus a Retry packet are defined. These packets use an 8-byte client-chosen Destination Connection ID of 0x8394c8f03e515708. Some intermediate values are included. All values are shown in hexadecimal.

A.1. Keys

The labels generated by the HKDF-Expand-Label function are:

client in: 00200f746c73313320636c69656e7420696e00

server in: 00200f746c7331332073657276657220696e00

quic key: 00100e746c733133207175696320 6b657900

quic iv: 000c0d746c7331332071756963206976 00

quic hp: 00100d746c7331332071756963206870 00

The initial secret is common:

```
initial_secret = HKDF-Extract(initial_salt, cid)
    = 524e374c6da8cf8b496f4bcb69678350
      7aafee6198b202b4bc823ebf7514a423
```

The secrets for protecting client packets are:

```
client_initial_secret
    = HKDF-Expand-Label(initial_secret, "client in", _, 32)
    = fda3953aecc040e48b34e27ef87de3a6
      098ecf0e38b7e032c5c57bcbd5975b84

key = HKDF-Expand-Label(client_initial_secret, "quic key", _, 16)
    = af7fd7efebd21878ff66811248983694

iv  = HKDF-Expand-Label(client_initial_secret, "quic iv", _, 12)
    = 8681359410a70bb9c92f0420

hp  = HKDF-Expand-Label(client_initial_secret, "quic hp", _, 16)
    = a980b8b4fb7d9fbc13e814c23164253d
```

The secrets for protecting server packets are:

```
server_initial_secret
    = HKDF-Expand-Label(initial_secret, "server in", _, 32)
    = 554366b81912ff90be41f17e80222130
      90ab17d8149179bcadf222f29ff2ddd5

key = HKDF-Expand-Label(server_initial_secret, "quic key", _, 16)
    = 5d51da9ee897a21b2659ccc7e5bfa577

iv  = HKDF-Expand-Label(server_initial_secret, "quic iv", _, 12)
    = 5e5ae651fd1e8495af13508b

hp  = HKDF-Expand-Label(server_initial_secret, "quic hp", _, 16)
    = a8ed82e6664f865aedf6106943f95fb8
```

A.2. Client Initial

The client sends an Initial packet. The unprotected payload of this packet contains the following CRYPTO frame, plus enough PADDING frames to make a 1162 byte payload:

```
060040c4010000c003036660261ff947 cea49cce6cfad687f457cf1b14531ba1
4131a0e8f309a1d0b9c4000006130113 031302010000910000000b0009000006
736572766572ff01000100000a001400 12001d00170018001901000101010201
030104002300000033002600240001d00 204cfdfcd178b784bf328cae793b136f
2aedce005ff183d7bb14952072366470 37002b0003020304000d0020001e0403
050306030203080408050806040105010 060102010402050206020202002d0002
0101001c00024001
```

The unprotected header includes the connection ID and a 4 byte packet number encoding for a packet number of 2:

```
c3ff000019088394c8f03e5157080000449e00000002
```

Protecting the payload produces output that is sampled for header protection. Because the header uses a 4 byte packet number encoding, the first 16 bytes of the protected payload is sampled, then applied to the header:

```
sample = 535064a4268a0d9d7b1c9d250ae35516

mask = AES-ECB(hp, sample)[0..4]
     = 833b343aaa

header[0] ^= mask[0] & 0x0f
     = c0
header[18..21] ^= mask[1..4]
     = 3b343aa8
header = c0ff000019088394c8f03e5157080000449e3b343aa8
```

The resulting protected packet is:

```
c0ff000019088394c8f03e5157080000 449e3b343aa8535064a4268a0d9d7b1c
9d250ae355162276e9b1e3011ef6bbc0 ab48ad5bcc2681e953857ca62becd752
4daac473e68d7405fbba4e9ee616c870 38bdbe908c06d9605d9ac49030359eec
b1d05a14e117db8cede2bb09d0dbbfee 271cb374d8f10abec82d0f59a1dee29f
e95638ed8dd41da07487468791b719c5 5c46968eb3b54680037102a28e53dc1d
12903db0af5821794b41c4a93357fa59 ce69cfe7f6bdfa629eef78616447e1d6
11c4baf71bf33febcb03137c2c75d253 17d3e13b684370f668411c0f00304b50
1c8fd422bd9b9ad81d643b20da89ca05 25d24d2b142041cae0af205092e43008
0cd8559ea4c5c6e4fa3f66082b7d303e 52ce0162baa958532b0bbc2bc785681f
cf37485dff6595e01e739c8ac9efba31 b985d5f656cc092432d781db95221724
87641c4d3ab8ece01e39bc85b1543661 4775a98ba8fa12d46f9b35e2a55eb72d
7f85181a366663387ddc20551807e007 673bd7e26bf9b29b5ab10a1ca87cbb7a
d97e99eb66959c2a9bc3cbde4707ff77 20b110fa95354674e395812e47a0ae53
b464dcb2d1f345df360dc227270c7506 76f6724eb479f0d2fbb6124429990457
ac6c9167f40aab739998f38b9eccb24f d47c8410131bf65a52af841275d5b3d1
880b197df2b5dea3e6de56ebce3ffb6e 9277a82082f8d9677a6767089b671ebd
244c214f0bde95c2beb02cd1172d58bd f39dce56ff68eb35ab39b49b4eac7c81
5ea60451d6e6ab82119118df02a58684 4a9ffe162ba006d0669ef57668cab38b
62f71a2523a084852cd1d079b3658dc2 f3e87949b550bab3e177cfc49ed190df
f0630e43077c30de8f6ae081537f1e83 da537da980afa668e7b7fb25301cf741
524be3c49884b42821f17552fbd1931a 813017b6b6590a41ea18b6ba49cd48a4
40bd9a3346a7623fb4ba34a3ee571e3c 731f35a7a3cf25b551a680fa68763507
b7fde3aaf023c50b9d22da6876ba337e b5e9dd9ec3daf970242b6c5aab3aa4b2
96ad8b9f6832f686ef70fa938b31b4e5 ddd7364442d3ea72e73d668fb0937796
f462923a81a47e1cee7426ff6d922126 9b5a62ec03d6ec94d12606cb485560ba
b574816009e96504249385bb61a819be 04f62c2066214d8360a2022beb316240
b6c7d78bbe56c13082e0ca272661210a bf020bf3b5783f1426436cf9ff418405
93a5d0638d32fc51c5c65ff291a3a7a5 2fd6775e623a4439cc08dd25582febc9
44ef92d8dbd329c91de3e9c9582e41f1 7f3d186f104ad3f90995116c682a2a14
a3b4b1f547c335f0be710fc9fc03e0e5 87b8cda31ce65b969878a4ad4283e6d5
b0373f43da86e9e0ffe1ae0fddd35162 55bd74566f36a38703d5f34249ded1f6
6b3d9b45b9af2ccfefe984e13376b1b2 c6404aa48c8026132343da3f3a33659e
c1b3e95080540b28b7f3fcd35fa5d843 b579a84c089121a60d8c1754915c344e
eaf45a9bf27dc0c1e784161691220913 13eb0e87555abd706626e557fc36a04f
cd191a58829104d6075c5594f627ca50 6bf181daec940f4a4f3af0074eee89da
acde6758312622d4fa675b39f728e062 d2bee680d8f41a597c262648bb18bcfc
13c8b3d97b1a77b2ac3af745d61a34cc 4709865bac824a94bb19058015e4e42d
aebe13f98ec51170a4aad0a8324bb768
```

A.3. Server Initial

The server sends the following payload in response, including an ACK frame, a CRYPTO frame, and no PADDING frames:

```
0d0000000018410a020000560303eefc e7f7b37ba1d1632e96677825ddf73988
cfc79825df566dc5430b9a045a120013 0100002e00330024001d00209d3c940d
89690b84d08a60993c144eca684d1081 287c834d5311bcf32bb9da1a002b0002
0304
```

The header from the server includes a new connection ID and a 2-byte packet number encoding for a packet number of 1:

```
c1ff0000190008f067a5502a4262b50040740001
```

As a result, after protection, the header protection sample is taken starting from the third protected octet:

```
sample = 7002596f99ae67abf65a5852f54f58c3
mask   = 38168a0c25
header = c9ff0000190008f067a5502a4262b5004074168b
```

The final protected packet is then:

```
c9ff0000190008f067a5502a4262b500 4074168bf22b7002596f99ae67abf65a
5852f54f58c37c808682e2e40492d8a3 899fb04fc0afe9aabc8767b18a0aa493
537426373b48d502214dd856d63b78ce e37bc664b3fe86d487ac7a77c53038a3
cd32f0b5004d9f5754c4f7f2d1f35cf3 f7116351c92b99c8ae5833225cb51855
20d61e68cf5f
```

A.4. Retry

This shows a Retry packet that might be sent in response to the Initial packet in Appendix A.2. The integrity check includes the client-chosen connection ID value of 0x8394c8f03e515708, but that value is not included in the final Retry packet:

```
ffff0000190008f067a5502a4262b574 6f6b656e1e5ec5b014cbb1f0fd93df40
48c446a6
```

Appendix B. 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.

B.1. Since draft-ietf-quic-tls-25

* No changes

B.2. Since draft-ietf-quic-tls-24

* Rewrite key updates (#3050)
  - Allow but don't recommend deferring key updates (#2792, #3263)
  - More completely define received behavior (#2791)
  - Define the label used with HKDF-Expand-Label (#3054)

B.3. Since draft-ietf-quic-tls-23

*Key update text update (#3050):

  -Recommend constant-time key replacement (#2792)

  -Provide explicit labels for key update key derivation (#3054)

*Allow first Initial from a client to span multiple packets (#2928, #3045)

*PING can be sent at any encryption level (#3034, #3035)

B.4. Since draft-ietf-quic-tls-22

*Update the salt used for Initial secrets (#2887, #2980)

B.5. Since draft-ietf-quic-tls-21

*No changes

B.6. Since draft-ietf-quic-tls-20

*Mandate the use of the QUIC transport parameters extension (#2528, #2560)

*Define handshake completion and confirmation; define clearer rules when it encryption keys should be discarded (#2214, #2267, #2673)

B.7. Since draft-ietf-quic-tls-18

*Increased the set of permissible frames in 0-RTT (#2344, #2355)

*Transport parameter extension is mandatory (#2528, #2560)

B.8. Since draft-ietf-quic-tls-17

*Endpoints discard initial keys as soon as handshake keys are available (#1951, #2045)

*Use of ALPN or equivalent is mandatory (#2263, #2284)

B.9. Since draft-ietf-quic-tls-14

*Update the salt used for Initial secrets (#1970)

*Clarify that TLS_AES_128_CCM_8_SHA256 isn't supported (#2019)

* Change header protection

  - Sample from a fixed offset (#1575, #2030)

  - Cover part of the first byte, including the key phase (#1322, #2006)

* TLS provides an AEAD and KDF function (#2046)

  - Clarify that the TLS KDF is used with TLS (#1997)

  - Change the labels for calculation of QUIC keys (#1845, #1971, #1991)

* Initial keys are discarded once Handshake keys are available (#1951, #2045)

B.10. Since draft-ietf-quic-tls-13

* Updated to TLS 1.3 final (#1660)

B.11. Since draft-ietf-quic-tls-12

* Changes to integration of the TLS handshake (#829, #1018, #1094, #1165, #1190, #1233, #1242, #1252, #1450)

  - The cryptographic handshake uses CRYPTO frames, not stream 0

  - QUIC packet protection is used in place of TLS record protection

  - Separate QUIC packet number spaces are used for the handshake

  - Changed Retry to be independent of the cryptographic handshake

  - Limit the use of HelloRetryRequest to address TLS needs (like key shares)

* Changed codepoint of TLS extension (#1395, #1402)

B.12. Since draft-ietf-quic-tls-11

* Encrypted packet numbers.

B.13. Since draft-ietf-quic-tls-10

* No significant changes.

B.14. Since draft-ietf-quic-tls-09

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

B.15. Since draft-ietf-quic-tls-08

- Specify value for max_early_data_size to enable 0-RTT (#942)

- Update key derivation function (#1003, #1004)

B.16. Since draft-ietf-quic-tls-07

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

B.17. Since draft-ietf-quic-tls-05

No significant changes.

B.18. Since draft-ietf-quic-tls-04

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

B.19. Since draft-ietf-quic-tls-03

No significant changes.

B.20. Since draft-ietf-quic-tls-02

- Updates to match changes in transport draft

B.21. 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)

B.22. 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

B.23. Since draft-thomson-quic-tls-01

*Adopted as base for draft-ietf-quic-tls

*Updated authors/editors list

*Added status note

Contributors

The IETF QUIC Working Group received an enormous amount of support from many people. The following people provided substantive contributions to this document: Adam Langley, Alessandro Ghedini, Christian Huitema, Christopher Wood, David Schinazi, Dragana Damjanovic, Eric Rescorla, Ian Swett, Jana Iyengar, (Kazuho Oku), Marten Seemann, Martin Duke, Mike Bishop, Mikkel Fahnøe Jørgensen, Nick Banks, Nick Harper, Roberto Peon, Rui Paulo, Ryan Hamilton, and Victor Vasiliev.

Authors' Addresses

Martin Thomson (editor)
Mozilla

Email: mt@lowentropy.net

Sean Turner (editor)
sn3rd

Email: sean@sn3rd.com