QUIC M. Thomson, Ed.
Internet-Draft Mozilla
Intended status: Standards Track S. Turner, Ed.
Expires: January 9, 2020 sn3rd
July 08, 2019

Using TLS to Secure QUIC
draft-ietf-quic-tls-21

Abstract

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

Note to Readers

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

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

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 below:

```
+--------------+--------------+--------------+
|  Handshake   |    Alerts    |  Application |
|    Layer     |              |     Data     |
|              |              |              |
+--------------+--------------+--------------+
|                                            |
|                Record Layer                |
|                                            |
+--------------------------------------------+
```

Each upper layer (handshake, alerts, and application data) is carried as a series of typed TLS records. Records are individually cryptographically protected and then transmitted over a reliable transport (typically TCP) which provides sequencing and guaranteed delivery.

Change Cipher Spec records cannot be sent in QUIC.

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

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

The TLS key exchange is 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 1. Note that this omits the EndOfEarlyData message, which is not used in QUIC (see Section 8.3).

```
       Client                                                  Server

       ClientHello
      (0-RTT Application Data)  -------->
                                                          ServerHello
                                                 {EncryptedExtensions}
                                                            {Finished}
                                <--------          [Application Data]
      {Finished}                -------->

      [Application Data]        <------->          [Application Data]

       () Indicates messages protected by early data (0-RTT) keys
       {} Indicates messages protected using handshake keys
       [] Indicates messages protected using application data
          (1-RTT) keys
```

Figure 1: TLS Handshake with 0-RTT

Data is protected using a number of encryption levels:

- o Initial Keys
- o Early Data (0-RTT) Keys
- o Handshake Keys
- o 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 below.

```
+--------------+--------------+ +-------------+
|     TLS      |     TLS      | |    QUIC     |
|  Handshake   |    Alerts    | | Applications|
|              |              | |  (h3, etc.) |
+--------------+--------------+-+-------------+
|                                             |
|                QUIC Transport               |
|   (streams, reliability, congestion, etc.)  |
|                                             |
+---------------------------------------------+
|                                             |
|            QUIC Packet Protection           |
|                                             |
+---------------------------------------------+
```

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 are co-dependent: 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:

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

- o 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 2 shows these interactions in more detail, with the QUIC packet protection being called out specially.

```
+------------+                        +------------+
|            |<- Handshake Messages ->|            |
|            |<---- 0-RTT Keys -------|            |
|            |<--- Handshake Keys-----|            |
|   QUIC     |<---- 1-RTT Keys -------|    TLS     |
|            |<--- Handshake Done ----|            |
+------------+                        +------------+
 |         ^
 | Protect | Protected
 v         | Packet
+------------+
|   QUIC     |
|  Packet    |
| Protection |
+------------+
```

Figure 2: 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 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:

o PADDING frames MAY appear in packets of any encryption level.

- CRYPTO and CONNECTION_CLOSE frames MAY appear in packets of any encryption level except 0-RTT.

- 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, 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
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 2, the interface from QUIC to TLS consists of three primary functions:

- Sending and receiving handshake messages

- 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 an endpoint when the following two conditions are met: the handshake is complete, and the endpoint has received an acknowledgment for a packet sent with 1-RTT keys. This second condition can be implemented by recording the lowest packet number sent with 1-RTT keys, and the highest value of the Largest Acknowledged field in any received 1-RTT ACK frame: once the latter is higher 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 given 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.

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 TLS starts using keys at a given encryption level, TLS indicates to QUIC that it is now reading or writing with keys at that encryption level. 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.

4.1.5. TLS Interface Summary

Figure 3 summarizes the exchange between QUIC and TLS for both client and server. Each arrow is tagged with the encryption level used for that transmission.

```
Client                                                    Server

Get Handshake
                     Initial ------------->
Install tx 0-RTT Keys
                     0-RTT --------------->
                                              Handshake Received
                                                   Get Handshake
                     <------------- Initial
                                           Install rx 0-RTT keys
                                          Install Handshake keys
                                                   Get Handshake
                     <----------- Handshake
                                           Install tx 1-RTT keys
                     <--------------- 1-RTT
Handshake Received
Install tx Handshake keys
Handshake Received
Get Handshake
Handshake Complete
                     Handshake ----------->
Install 1-RTT keys
                     1-RTT --------------->
                                              Handshake Received
                                           Install rx 1-RTT keys
                                              Handshake Complete
                                                   Get Handshake
                     <--------------- 1-RTT
Handshake Received
```

Figure 3: Interaction Summary between QUIC and TLS

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

QUIC requires that the first Initial packet from a client contain an entire cryptographic handshake message, which for TLS is the ClientHello. Though a packet larger than 1200 bytes might be supported by the path, a client improves the likelihood that a packet is accepted if it ensures that the first ClientHello message is small enough to stay within this limit.

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. Minimizing the size of these values increases the probability that they can be successfully used by a client.

A client is not required to fit the ClientHello that it sends in response to a HelloRetryRequest message into a single UDP datagram.

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 (see Section 4.6.2 of [TLS13]).

4.5. Enabling 0-RTT

In order to be usable for 0-RTT, TLS MUST provide 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. 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.

4.6. Rejecting 0-RTT

A server rejects 0-RTT by rejecting 0-RTT at the TLS layer. This also prevents QUIC from sending 0-RTT data. A server will always reject 0-RTT if it sends a TLS HelloRetryRequest.

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. 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.

4.8. 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.

4.9. 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.

4.9.1. 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.

4.9.2. Discarding Handshake Keys

An endpoint MUST NOT discard its handshake keys until the TLS handshake is confirmed (Section 4.1.2). An endpoint SHOULD discard its handshake keys as soon as it has confirmed the handshake. Most application protocols will send data after the handshake, resulting in acknowledgements that allow both endpoints to discard their handshake keys promptly. Endpoints that do not have reason to send immediately after completing the handshake MAY send ack-eliciting frames, such as PING, which will cause the handshake to be confirmed when they are acknowledged.

4.9.3. 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 first Initial packet of the connection. Specifically:

```
initial_salt = 0x7fbcdb0e7c66bbe9193a96cd21519ebd7a02644a
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.

Appendix A contains test vectors for the initial packet encryption.

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.

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 4 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 4: Header Protection Pseudocode

Figure 5 shows the protected fields of long and short headers marked with an E. Figure 5 also shows the sampled fields.

```
Long Header:
+-+-+-+-+-+-+-+-+
|1|1|T T|E E E E|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Version -> Length Fields                 ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Short Header:
+-+-+-+-+-+-+-+-+
|0|1|S|E E E E E|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               Destination Connection ID (0/32..144)         ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Common Fields:
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|E E E E E E E E E  Packet Number (8/16/24/32) E E E E E E E E...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   [Protected Payload (8/16/24)]             ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Sampled part of Protected Payload (128)         ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 Protected Payload Remainder (*)             ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 5: 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 = 6 + 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 interpreted as a 32-bit number in little-endian order and are used as the block count. The remaining 12 bytes are interpreted as three concatenated 32-bit numbers in little-endian order and used as the nonce.

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

```
counter = DecodeLE(sample[0..3])
nonce = DecodeLE(sample[4..7], sample[8..11], sample[12..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.

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 is limited before the handshake is complete. A server MUST NOT process data from 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.

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.

6. Key Update

Once the handshake is confirmed, it is possible to update the keys. The KEY_PHASE bit in the short header is used to indicate whether key updates have occurred. The KEY_PHASE bit is initially set to 0 and then inverted with each key update.

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.

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 4.8).

An endpoint MUST NOT initiate the first key update until the handshake is confirmed (Section 4.1.2). An endpoint MUST NOT initiate a subsequent key update until it has received an acknowledgment for a packet sent at the current KEY_PHASE. 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.

Endpoints MAY limit the number of keys they retain to two sets for removing packet protection and one set for protecting packets. Older keys can be discarded. Updating keys multiple times rapidly can cause packets to be effectively lost if packets are significantly reordered. Therefore, an endpoint SHOULD NOT initiate a key update for some time after it has last updated keys; the RECOMMENDED time period is three times the PTO. This avoids valid reordered packets being dropped by the peer as a result of the peer discarding older keys.

A receiving endpoint detects an update when the KEY_PHASE bit does not match what it is expecting. It creates a new secret (see Section 7.2 of [TLS13]) and the corresponding read key and IV using the KDF function provided by TLS. The header protection key is not updated.

If the packet can be decrypted and authenticated using the updated key and IV, then the keys the endpoint uses for packet protection are also updated. The next packet sent by the endpoint MUST then use the new keys. Once an endpoint has sent a packet encrypted with a given key phase, it MUST NOT send a packet encrypted with an older key phase.

An endpoint does not always need to send packets 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 period of no more than three times the PTO. After this period, old keys and their corresponding

secrets SHOULD be discarded. Retaining keys allow endpoints to process packets that were sent with old keys and delayed in the network. Packets with higher packet numbers always use the updated keys and MUST NOT be decrypted with old keys.

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.

```
   Initiating Peer                    Responding Peer

@M QUIC Frames
               New Keys -> @N
@N QUIC Frames
                      -------->
                                          QUIC Frames @M
                          New Keys -> @N
                                          QUIC Frames @N
                      <--------
```

Figure 6: Key Update

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

In deciding when to update keys, endpoints MUST NOT exceed the limits for use of specific keys, as described in Section 5.5 of [TLS13].

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 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 Negotiation

QUIC requires that the cryptographic handshake provide authenticated protocol negotiation. TLS uses Application Layer Protocol Negotiation (ALPN) [RFC7301] 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 4.8). While [RFC7301] only specifies that servers use this alert, QUIC clients MUST also use it to terminate a connection when ALPN negotiation fails.

An application-layer protocol MAY restrict the QUIC versions that it can operate over. Servers MUST select an application protocol compatible with the QUIC version that the client has selected. If the server cannot select a compatible combination of application protocol and QUIC version, it MUST abort the connection. A client MUST abort a connection if the server picks an incompatible combination of QUIC version and ALPN identifier.

8.2. QUIC Transport Parameters Extension

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

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

```
enum {
   quic_transport_parameters(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 carries a TransportParameters struct when the version of QUIC defined in [QUIC-TRANSPORT] is used.

The quic_transport_parameters extension is carried in the ClientHello and the EncryptedExtensions messages during the handshake. 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 4.8).

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. Peer Denial of Service

QUIC, TLS, and HTTP/2 all contain 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.

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.

9.4. Header Protection Analysis

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

The header protection algorithms defined in this document take the form:

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

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

Use of the same key and ciphertext sample more than once risks compromising 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.

An attacker could guess values for packet numbers 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.

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:

o TLS ExtensionsType 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.

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)", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.197, November 2001.

[CHACHA] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 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", draft-ietf-quic-recovery-21 (work in progress), July 2019.

[QUIC-TRANSPORT] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-21 (work in progress), July 2019.

[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>.

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

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

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

[TLS-REGISTRIES] Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", 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", March 2016, <http://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf>.

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

[QUIC-HTTP] Bishop, M., Ed., "Hypertext Transfer Protocol (HTTP) over QUIC", draft-ietf-quic-http-21 (work in progress), July 2019.

[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>.

11.3. URIs

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

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

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

Appendix A. Sample Initial Packet Protection

This section shows examples of packet protection for Initial packets so that implementations can be verified incrementally. These packets use an 8-byte client-chosen Destination Connection ID of 0x8394c8f03e515708. Values for both server and client packet protection are shown together with values in hexadecimal.

A.1. Keys

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

client in: 00200f746c73313320636c69656e7420696e00

server in: 00200f746c7331332073657276657220696e00

quic key: 00100e746c7331332071756963206b657900

quic iv: 000c0d746c733133207175696320697600

quic hp: 00100d746c733133207175696320687000

The initial secret is common:

```
initial_secret = HKDF-Extract(initial_salt, cid)
    = 4496d3903d3f97cc5e45ac5790ddc686
      683c7c0067012bb09d900cc21832d596
```

The secrets for protecting client packets are:

```
client_initial_secret
    = HKDF-Expand-Label(initial_secret, "client in", _, 32)
    = 8a3515a14ae3c31b9c2d6d5bc58538ca
      5cd2baa119087143e60887428dcb52f6

key = HKDF-Expand-Label(client_initial_secret, "quic key", _, 16)
    = 98b0d7e5e7a402c67c33f350fa65ea54

iv  = HKDF-Expand-Label(client_initial_secret, "quic iv", _, 12)
    = 19e94387805eb0b46c03a788

hp  = HKDF-Expand-Label(client_initial_secret, "quic hp", _, 16)
    = 0edd982a6ac527f2eddcbb7348dea5d7
```

The secrets for protecting server packets are:

```
server_initial_secret
    = HKDF-Expand-Label(initial_secret, "server in", _, 32)
    = 47b2eaea6c266e32c0697a9e2a898bdf
      5c4fb3e5ac34f0e549bf2c58581a3811

key = HKDF-Expand-Label(server_initial_secret, "quic key", _, 16)
    = 9a8be902a9bdd91d16064ca118045fb4

iv  = HKDF-Expand-Label(server_initial_secret, "quic iv", _, 12)
    = 0a82086d32205ba22241d8dc

hp  = HKDF-Expand-Label(server_initial_secret, "quic hp", _, 16)
    = 94b9452d2b3c7c7f6da7fdd8593537fd
```

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 an 1163 byte payload:

```
060040c4010000c003036660261ff947 cea49cce6cfad687f457cf1b14531ba1
4131a0e8f309a1d0b9c4000006130113 031302010000910000000b0009000006
736572766572ff01000100000a001400 12001d00170018001901000101010201
03010400230000003300260024001d00 204cfdfcd178b784bf328cae793b136f
2aedce005ff183d7bb14952072366470 37002b0003020304000d0020001e0403
05030603020308040805080604010501 060102010402050206020202002d0002
0101001c00024001
```

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

```
c3ff000015508394c8f03e51570800449f00000002
```

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 = 65f354ebb400418b614f73765009c016

mask = AES-ECB(hp, sample)[0..4]
     = 519bd343ff

header[0] ^= mask[0] & 0x0f
     = c2
header[17..20] ^= mask[1..4]
     = 9bd343fd
header = c2ff000015508394c8f03e51570800449f9bd343fd
```

The resulting protected packet is:

```
c2ff000015508394c8f03e5157080044 9f9bd343fd65f354ebb400418b614f73
765009c0162d594777f9e6ddeb32fba3 865cffd7e26e3724d4997cdde8df34f8
868772fed2412d43046f44dc7c6adf5e e10da456d56c892c8f69594594e8dcab
edb10d591130ca464588f2834eab931b 10feb963c1947a05f57062692c242248
ad0133b31f6dcc585ba344ca5beb382f b619272e65dfccae59c08eb00b7d2a5b
bccd888582df1d1aee040aea76ab4dfd cae126791e71561b1f58312edb31c164
ff1341fd2820e2399946bad901e425da e58a9859ef1825e7d757a6291d9ba6ee
1a8c836dc0027cd705bd2bc67f56bad0 024efaa3819cbb5d46cefdb7e0df3ad9
2b0689650e2b49ac29e6398bedc75554 1a3f3865bc4759bec74d721a28a0452c
1260189e8e92f844c91b27a00fc5ed6d 14d8fceb5a848bea0a3208162c7a9578
2fcf9a045b20b76710a2565372f25411 81030e4350e199e62fa4e2e0bba19ff6
6662ab8cc6815eeaa20b80d5f31c41e5 51f558d2c836a215ccff4e8afd2fec4b
fcb9ea9d051d12162f1b14842489b69d 72a307d9144fced64fc4aa21ebd310f8
97cf00062e90dad5dbf04186622e6c12 96d388176585fdb395358ecfec4d95db
4429f4473a76210866fd180eaeb60da4 33500c74c00aef24d77eae81755faa03
e71a8879937b32d31be2ba51d41b5d7a 1fbb4d952b10dd2d6ec171a3187cf3f6
4d520afad796e4188bc32d153241c083 f225b6e6b845ce9911bd3fe1eb4737b7
1c8d55e3962871b73657b1e2cce368c7 400658d47cfd9290ed16cdc2a6e3e7dc
ea77fb5c6459303a32d58f62969d8f46 70ce27f591c7a59cc3e7556eda4c58a3
2e9f53fd7f9d60a9c05cd6238c71e3c8 2d2efabd3b5177670b8d595151d7eb44
aa401fe3b5b87bdb88dffb2bfb6d1d0d 8868a41ba96265ca7a68d06fc0b74bcc
ac55b038f8362b84d47f52744323d08b 46bfec8c421f991e1394938a546a7482
a17c72be109ea4b0c71abc7d9c0ac096 0327754e1043f18a32b9fb402fc33fdc
b6a0b4fdbbddbdf0d85779879e98ef21 1d104a5271f22823f16942cfa8ace68d
0c9e5b52297da9702d8f1de24bcd0628 4ac8aa1068fa21a82abbca7e7454b848
d7de8c3d43560541a362ff4f6be06c01 15e3a733bff44417da11ae668857bba2
c53ba17db8c100f1b5c7c9ea960d3f3d 3b9e77c16c31a222b498a7384e286b9b
7c45167d5703de715f9b06708403562d cff77fdf2793f94e294888cebe8da4ee
88a53e38f2430addc161e8b2e2f2d405 41d10cda9a7aa518ac14d0195d8c2012
0b4f1d47d6d0909e69c4a0e641b83c1a d4fff85af4751035bc5698b6141ecc3f
bffcf2f55036880071ba118927400796 7f64468172854d140d229320d689f576
60f6c445e629d15ff2dcdff4b71a41ec 0c24bd2fd8f5ad13b2c3688e0fdb8dbc
ce42e6cf49cf60d022ccd5b19b4fd5d9 8dc10d9ce3a626851b1fdd23e1fa3a96
1f9b0333ab8d632e48c944b82bdd9e80 0fa2b2b9e31e96aee54b40edaf6b79ec
211fdc95d95ef552aa532583d76a539e 988e416a0a10df2550cdeacafc3d61b0
b0a79337960a0be8cf6169e4d55fa6e7 a9c2e8efabab3da008f5bcc38c1bbabd
b6c10368723da0ae83c4b1819ff54946 e7806458d80d7be2c867d46fe1f029c5
e952eb19ded16fabb19980480eb0fbcd
```


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:

```
c1ff00001505f067a5502a4262b50040740001
```

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

```
sample = 6176fa3b713f272a9bf03ee28d3c8add
mask   = 5bd74a846c
header = caff00001505f067a5502a4262b5004074d74b
```

The final protected packet is then:

```
caff00001505f067a5502a4262b50040 74d74b7e486176fa3b713f272a9bf03e
e28d3c8addb4e805b3a110b663122a75 eee93c9177ac6b7a6b548e15a7b8f884
65e9eab253a760779b2e6a2c574882b4 8d3a3eed696e50d04d5ec59af85261e4
cdbe264bd65f2b076760c69beef23aa7 14c9a174d69034c09a2863e1e1863508
8d4afdeab9
```

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-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.2. 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.3. 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.4. 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 are avaialble (#1951, #2045)

B.5. Since draft-ietf-quic-tls-13

- Updated to TLS 1.3 final (#1660)

B.6. 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.7. Since draft-ietf-quic-tls-11

- Encrypted packet numbers.

B.8. Since draft-ietf-quic-tls-10

- No significant changes.

B.9. Since draft-ietf-quic-tls-09

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

B.10. 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.11. Since draft-ietf-quic-tls-07

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

B.12. Since draft-ietf-quic-tls-05

No significant changes.

B.13. Since draft-ietf-quic-tls-04

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

B.14. Since draft-ietf-quic-tls-03

No significant changes.

B.15. Since draft-ietf-quic-tls-02

- Updates to match changes in transport draft

B.16. 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.17. 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.18. Since draft-thomson-quic-tls-01

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

- Updated authors/editors list

- Added status note

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.

Contributors

Ryan Hamilton was originally an author of this specification.

Authors' Addresses

Martin Thomson (editor)
Mozilla

Email: mt@lowentropy.net

Sean Turner (editor)
sn3rd

Email: sean@sn3rd.com