QUIC
Internet-Draft
Intended status: Standards Track
Expires: February 16, 2019

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

August 15, 2018

Using Transport Layer Security (TLS) to Secure QUIC
draft-ietf-quic-tls-14

Abstract

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

Note to Readers

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

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

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

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

This document describes how the standardized TLS 1.3 acts 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.

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.

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

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

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

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

o 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.

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

A simplified TLS 1.3 handshake with 0-RTT application data is shown in Figure 1, see [TLS13] for more options and details.

```
    Client                                             Server

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

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 1.3 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|
|              |              | | (h2q, etc.) |
+--------------+--------------+-+-------------+
|                                             |
|                QUIC Transport               |
|   (streams, reliability, congestion, etc.)  |
|                                             |
+---------------------------------------------+
|                                             |
|            QUIC Packet Protection           |
|                                             |
+---------------------------------------------+
```

QUIC also relies on TLS 1.3 for authentication and negotiation of parameters that are critical to security and performance.

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

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 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 1.3 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.

Each encryption level has a specific list of frames which may appear in it. 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

- CRYPTO frames MAY appear in packets of any encryption level.

- CONNECTION_CLOSE MAY appear in packets of any encryption level other than 0-RTT.

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

- STREAM frames MUST ONLY appear in the 0-RTT and 1-RTT levels.

- All other frame types MUST only appear at the 1-RTT levels.

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
Short Header	1-RTT	0/1-RTT

Table 1: Encryption Levels by Packet Type

Section 6.5 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:

o Sending and receiving handshake messages

o Rekeying (both transmit and receive)

o Handshake state updates

Additional functions might be needed to configure TLS.

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

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 octets 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.

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

o 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.

o 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 octets are requested from TLS. TLS might not provide any octets if the handshake messages it has received are incomplete or it has no data to send.

Once the TLS handshake is complete, this is indicated to QUIC along with any final handshake octets that TLS needs to send. 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.

Important: Until the handshake is reported as complete, the connection and key exchange are not properly authenticated at the server. Even though 1-RTT keys are available to a server after receiving the first handshake messages from a client, the server cannot consider the client to be authenticated until it receives and validates the client's Finished message.

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

4.1.2. 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 octets, or after TLS produces handshake octets.

If 0-RTT is possible, it is ready after the client sends a TLS ClientHello message or the server receives that message. After providing a QUIC client with the first handshake octets, the TLS stack might signal the change to 0-RTT keys. On the server, after receiving handshake octets 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 before the final TLS handshake message at the 0-RTT encryption level (EndOfEarlyData) is available. Though the content of CRYPTO frames at the Handshake encryption level cannot be forwarded to TLS before EndOfEarlyData is processed, the client could send ACK frames that the server needs to process in order to detect lost Handshake packets.

4.1.3. 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 ------------->
   Rekey tx to 0-RTT Keys
                        0-RTT --------------->
                                                 Handshake Received
                                                      Get Handshake
                        <------------- Initial
                                              Rekey rx to 0-RTT keys
                                                 Handshake Received
                                          Rekey rx to Handshake keys
                                                      Get Handshake
                        <----------- Handshake
                                              Rekey tx to 1-RTT keys
                        <--------------- 1-RTT
   Handshake Received
   Rekey rx to Handshake keys
   Handshake Received
   Get Handshake
   Handshake Complete
                        Handshake ----------->
   Rekey tx to 1-RTT keys
                        1-RTT --------------->
                                                 Handshake Received
                                              Rekey rx to 1-RTT keys
                                                      Get Handshake
                                                 Handshake Complete
                        <--------------- 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 crytographic handshake message, which for TLS is the ClientHello. Though a packet larger than 1200 octets 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 octets 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 octets, both of which might be required if a server sends a Retry packet.

A typical TLS ClientHello can easily fit into a 1200 octet 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 ID 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 "max_early_data" extension with the value 0xffffffff; the amount of data which the client can send in 0-RTT is controlled by the "initial_max_data" transport parameter supplied by the server. A client MUST treat receipt of a NewSessionTicket that contains a "max_early_data" extension with any other value as a connection error of type PROTOCOL_VIOLATION.

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

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 4.4 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-octet 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 availble. 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.

After all CRYPTO frames for a given encryption level have been sent and all expected CRYPTO frames received, and all the corresponding acknowledgments have been received or sent, an endpoint starts a timer. To limit the effect of packet loss around a change in keys, endpoints MUST retain packet protection keys for that encryption level for at least three times the current Retramsmission Timeout (RTO) interval as defined in [QUIC-RECOVERY]. Retaining keys for this interval allows packets containing CRYPTO or ACK frames at that encryption level to be sent if packets are determined to be lost or new packets require acknowledgment.

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.

Once this timer expires, an endpoint MUST NOT either accept or generate new packets using those packet protection keys. An endpoint can discard packet protection keys for that encryption level.

Key updates (see Section 6) can be used to update 1-RTT keys before keys from other encryption levels are discarded. In that case, packets protected with the newest packet protection keys and packets sent two updates prior will appear to use the same keys. After the handshake is complete, endpoints only need to maintain the two latest sets of packet protection keys and MAY discard older keys. Updating keys multiple times rapidly can cause packets to be effectively lost if packets are significantly delayed. Because key updates can only be performed once per round trip time, only packets that are delayed by more than a round trip will be lost as a result of changing keys; such packets will be marked as lost before this, as they leave a gap in the sequence of packet numbers.

5. QUIC Packet Protection

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

5.1. QUIC Packet Encryption Keys

QUIC derives packet encryption keys in the same way as TLS 1.3: Each encryption level/direction pair has a secret value, which is then used to derive the traffic keys using as described in Section 7.3 of [TLS13]

The keys for the Initial encryption level are computed based on the client's initial Destination Connection ID, as described in Section 5.1.1.

The keys for other encryption levels are computed in the same fashion as the corresponding TLS keys (see Section 7 of [TLS13]), except that the label for HKDF-Expand-Label uses the prefix "quic " rather than "tls13 ". A different label provides key separation between TLS and QUIC.

5.1.1. 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 = 0x9c108f98520a5c5c32968e950e8a2c5fe06d6c38
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)
```

Note that if the server sends a Retry, the client's Initial will correspond to a new connection and thus use the server provided Destination Connection ID.

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 initial Destination Connection ID.

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

Note: The Destination Connection ID is of arbitrary length, and it could be zero length if the server sends a Retry packet with a zero-length Source Connection ID field. In this case, the 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.

5.2. QUIC 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.

QUIC packets are protected prior to applying packet number encryption (Section 5.3). The unprotected packet number is part of the associated data (A). When removing packet protection, an endpoint first removes the protection from the packet number.

All QUIC packets other than Version Negotiation and 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 5.1.1). This provides protection against off-path attackers and robustness against QUIC version unaware middleboxes, but not against on-path attackers.

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

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 64 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 octet in either the short or long header, up to and including the unprotected packet number.

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

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

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.

5.3. Packet Number Protection

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

Packet number protection is applied after packet protection is applied (see Section 5.2). The ciphertext of the packet is sampled and used as input to an encryption algorithm.

In sampling the packet ciphertext, the packet number length is assumed to be 4 octets (its maximum possible encoded length), unless there is insufficient space in the packet for sampling. The sampled ciphertext starts after allowing for a 4 octet packet number unless this would cause the sample to extend past the end of the packet. If the sample would extend past the end of the packet, the end of the packet is sampled.

For example, the sampled ciphertext for a packet with a short header can be determined by:

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

if sample_offset + sample_length > packet_length then
    sample_offset = packet_length - sample_length
sample = packet[sample_offset..sample_offset+sample_length]
```

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)
```

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

Packet number protection is applied to the packet number encoded as described in Section 4.11 of [QUIC-TRANSPORT]. Since the length of the packet number is stored in the first octet of the encoded packet number, it may be necessary to progressively decrypt the packet number.

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

5.3.1. AES-Based Packet Number Protection

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

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

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

5.3.2. ChaCha20-Based Packet Number Protection

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

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

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

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

5.4. 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.

5.5. 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. Clients MUST NOT attempt to decrypt 0-RTT packets it receives and instead MUST discard them.

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.

5.6. 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.

However, a server MUST NOT process data from incoming 1-RTT protected packets before verifying either the client Finished message or - in the case that the server has chosen to use a pre-shared key - the pre-shared key binder (see Section 4.2.11 of [TLS13]). Verifying these values provides the server with an assurance that the ClientHello has not been modified. Packets protected with 1-RTT keys MAY be stored and later decrypted and used once the handshake is complete.

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 1-RTT keys are established and the short header is in use, 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 Section 6.

The KEY_PHASE bit allows a recipient to detect a change in keying material without necessarily needing to receive the first packet that triggered the change. An endpoint that notices a changed KEY_PHASE bit can update keys and decrypt the packet that contains the changed bit, see Section 6.

An endpoint MUST NOT initiate more than one key update at a time. A new key cannot be used until the endpoint has received and successfully decrypted a packet with a matching KEY_PHASE.

A receiving endpoint detects an update when the KEY_PHASE bit doesn't match what it is expecting. It creates a new secret (see Section 7.2 of [TLS13]) and the corresponding read key and IV. If the packet can be decrypted and authenticated using these values, then the keys it uses for packet protection are also updated. The next packet sent by the endpoint will then use the new keys.

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

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

Keys and their corresponding secrets SHOULD be discarded when an endpoint has received all packets with packet numbers lower than the lowest packet number used for the new key. An endpoint might discard

keys if it determines that the length of the delay to affected packets is excessive.

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

```
   Initiating Peer                    Responding Peer

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

Figure 4: Key Update

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

7. **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 and Version Negotiation

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

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

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

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

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 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.

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.

9. Security Considerations

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

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

9.1. Packet Reflection Attack Mitigation

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

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 4.7 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.

9.2. Peer Denial of Service

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

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

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.

9.3. Packet Number Protection Analysis

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

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

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

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

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

To prevent an attacker from modifying packet numbers, values of packet numbers are transitively authenticated using packet protection; packet numbers are part of the authenticated additional data. A falsified or modified packet number can only be detected once the packet protection is removed.

An attacker can guess values for packet numbers and have an endpoint confirm guesses through timing side channels. If the recipient of a packet discards packets with duplicate packet numbers without attempting to remove packet protection they could reveal through timing side-channels that the packet number matches a received

packet. For authentication to be free from side-channels, the entire process of packet number protection removal, packet number recovery, and packet protection removal MUST be applied together without timing and other side-channels.

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

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-14 (work in progress), August 2018.

[QUIC-TRANSPORT] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-14 (work in progress), August 2018.

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

[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 Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", draft-ietf-tls-iana-registry-updates-05 (work in progress), May 2018.

[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-14 (work in progress), August 2018.

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

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

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. 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.

A.1. Since draft-ietf-quic-tls-13

o Updated to TLS 1.3 final (#1660)

A.2. Since draft-ietf-quic-tls-12

o 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)

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

A.3. Since draft-ietf-quic-tls-11

- Encrypted packet numbers.

A.4. Since draft-ietf-quic-tls-10

- No significant changes.

A.5. Since draft-ietf-quic-tls-09

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

A.6. Since draft-ietf-quic-tls-08

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

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

A.7. Since draft-ietf-quic-tls-07

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

A.8. Since draft-ietf-quic-tls-05

No significant changes.

A.9. Since draft-ietf-quic-tls-04

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

A.10. Since draft-ietf-quic-tls-03

No significant changes.

A.11. Since draft-ietf-quic-tls-02

- Updates to match changes in transport draft

A.12. 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)

A.13. 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

A.14. 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: martin.thomson@gmail.com

Sean Turner (editor)
sn3rd

Email: sean@sn3rd.com