QUIC Crypto

Adam Langley <agl@google.com>
Wan-Teh Chang <wtc@google.com>

(Revision 20130620.)

Summary
Source address spoofing
Replay protection
Handshake costs
Strike-register design
Wire Protocol
Client handshake.
Key derivation
Client encrypted tag values
Certificate compression
Future directions
Acknowledgements

Summary

The QUIC crypto protocol is the part of QUIC that provides transport security to a connection. When the client has cached information about the server, it can establish an encrypted connection with no round trips. TLS, in contrast, requires at least two round trips (counting the TCP 3-way handshake). QUIC handshakes should be ~5 times more efficient than common TLS handshakes (2048-bit RSA) and at a greater security level.

Source address spoofing

Very few protocols on the Internet work without at least one initialisation round trip. Most protocols have a round trip due to TCP and TLS-based protocols additionally have at least one more before application data can flow.

Both of these rounds trips exchange nonces: sequence numbers (or SYN cookies) in the case of TCP and cryptographic random values (`client_random` and `server_random`) in TLS. The TCP nonce prevents IP address spoofing and the TLS nonce prevents replay attacks. Any protocol which seeks to eliminate round trips has to somehow address these two problems.

As a counterexample, DNS is a protocol that doesn't have any initialisation round trips, thus it has to deal with IP address spoofing and replay attacks itself. DNS simply ignores IP address spoofing and

thus mirror DDoS attacks are a real problem. For replay protection, DNSSEC relies on clock synchronisation and short-lived signatures. This allows replays for a limited time by design because that meshes with DNS's caching semantics. But it's not a strict form of replay protection because replays are permitted.

In QUIC we deal with the two problems separately.

The IP address spoofing problem is handled by issuing the client, on demand, a "source-address token". This is an opaque byte string from the client's point of view. From the server's point of view it's an authenticated-encryption block (e.g. AES-GCM) that contains, at least, the client's IP address and a timestamp by the server. The server will only send a source address token for a given IP to that IP. Receipt of the token by the client is taken as proof of ownership of the IP address in the same way that receipt of a TCP sequence number is.

Clients can include the source address token in future requests in order to demonstrate ownership of their source IP address. If the client has moved IP addresses, the token is too old, or the client doesn't have a token, then the server may reject the connection and return a fresh token to the client. But if the client has remained on the same IP address then it can reuse a source-address token to avoid the round trip needed to obtain a fresh one.

The lifetime of a token is a matter for the server but, since source address tokens are bearer tokens, they can be stolen and reused in order to bypass IP address based restrictions. (Although the attacker would not receive the response.) Source address tokens can also be collected and possibly used after ownership of the IP address has changed (i.e. in a DHCP pool). Reducing the lifetime of tokens ameliorates both of these concerns at the cost of reducing the number of requests that can be handled without additional round trips.

Source address tokens, unlike an exchange of TCP sequence numbers, do not require the source to demonstrate a continuous ability to receive packets sent to the source IP address. This allows a source address token to be used to continuously request traffic from a server even once the downstream link has been saturated and the drop rate is high enough that TCP connections couldn't be established. This "self DOS" attack can be used to DOS other users on the same downstream links.

However, we note that a similar trick is actually possible with TCP and so QUIC doesn't make things obviously worse in this respect. Indeed, once the connection is established, QUIC includes an entropy bit in packets and requires that receivers send a hash of the entropies that they claim to have received - thus solving the issue with TCP.

In order to minimise latency, servers may decide to relax source address restrictions dynamically. One can imagine a server that tracks the number of requests coming from different IP addresses and only demands source-address tokens when the count of "unrequited" connections exceeds a limit globally, or for a certain IP range. This may well be effective but it's unclear whether this is globally stable. If a large number of QUIC servers implemented this strategy then a substantial mirror DDoS

attack may be split across them such that the attack threshold wasn't reached by any one server.

Replay protection

In TLS, each side generates an nonce which is used to ensure that the other party is fresh by forcing them to include the (assumed to be unique for all time) value in the key derivation. Without a round trip the client can still include an nonce and so ensure that the server is fresh, but the server doesn't have a chance to do so for the client. Therefore the server must do something else to prevent non-idempotent client actions being repeated by an attacker.

Since the server cannot introduce an nonce into the handshake without a round trip, the client must give a unique ID to every connection and the server must ensure that it never processes the same connection twice. The client's nonce serves as a suitable ID.

The requirement that the server never process the same connection twice is troublesome because it suggests that all the servers, worldwide, which serve a given domain, need to access a consistent, eternal register of previous client nonces. However, we can significantly relax that requirement by partitioning the state and by requiring clock sync.

We can partition the state by having the server ensure not that the client nonce is unique, but rather that the nonce is unique within a given "orbit". An orbit partitions the nonce space and identifies a set of servers which manage a shared register of previously observed client nonces. Servers reject connections where they don't recognise the orbit, thus servers don't need to keep track of those nonces. This scheme allows us to spatially partition the storage for large services. Clients cache the orbit value of the server and, if their handshake is routed to a different "orbit" of servers, then the handshake will need an extra round trip. Thus this scheme assumes the common practice of geographical routing of requests and colocated servers.

The register of client nonces conceptually "strikes off" each client nonce that is used, thus this register is referred to as the "strike-register".

The second way by which we can make the server's job easier is by requiring clock sync between the server and the client. If we mirror TLS's method of including a timestamp in the nonce then servers need only maintain state for a limited amount of time. Without this, servers would have to maintain the strike-register indefinitely.

Therefore, when accepting a new connection, the server must verify that the connection is fresh by checking to see whether the (client nonce, orbit) pair is contained in the strike-register. There are four possible cases:

1. The timestamp in the nonce is either before the starting validity of the register, or is too far into the future to store in the register.

2. The orbit isn't known.
3. The connection is in the strike-register
4. None of the above.

In the first two cases, the connection may be fresh, but the server must act conservatively and reject it as a replay. In the third case, the connection is clearly a replay. In the final case, the connection can be accepted and the register must be (atomically) updated.

Handshake costs

In TLS, the server picks connection parameters for each connection based on the client's advertised support for them. In QUIC, the server's preferences are fully enumerated and static. They are bundled, along with Diffie-Hellman public values into a "server config". This server config has an expiry and is signed by the server's private key. Because the server config is static, a signing operation is not needed for each connection, rather a single signature suffices for many connections.

The keys for a connection are agreed using Diffie-Hellman. The server's Diffie-Hellman value is found in the server config and the client provides one in its first handshake message. Because the server config must be kept for some time in order to allow 0-RTT handshakes, this puts an upper bound on the forward security of the connection. As long as the server keeps the Diffie-Hellman secrets for a server config, data encrypted using that server config could be decrypted if they leak.

Thus QUIC provides two levels of secrecy: the initial data from the client is encrypted using the Diffie-Hellman value in the server's server config, which may persist for several days. Immediately upon receiving the connection, the server replies with an ephemeral Diffie-Hellman value and the connection is rekeyed.

(This may appear to provide less forward security than a forward-secure TLS connection. However, to avoid round trips TLS SessionTickets are typically enabled in large deployments. The SessionTicket key is sufficient to decrypt a connection but, for resumption to be effective, it must be retained for reasonable period - typically some number of days. Thus the SessionTicket key and the server config key are analogous and the effective security is actually greater with QUIC because its forward-secure mode is then superior.)

A single connection is the usual scope for forward security, but the security difference between an ephemeral key used for a single connection, and one used for all connections for 60 seconds is negligible. Thus we can amortise the Diffie-Hellman key generation at the server over all the connections in a small time span.

(Because the server config and Diffie-Hellman private values are all the server needs in order to process QUIC connections, the private key for the certificate need never be placed on the server. Rather, a form of short-lived certificates can be implemented by signing short-lived server configs

and installing only those on the server.)

If we let **S** be a secret key operation (i.e. RSA decrypt), **P** be a public key operation (i.e. RSA encrypt), **F** be a Diffie-Hellman, fixed point, scalar multiplication and **A** be an arbitrary point, scalar multiplication then:

1. TLS, non-forward-secure handshake: server, 1**S** (1100µs); client, 1**P** (34µs).
2. TLS, forward-secure handshake: server, 1**S** + 1**F** + 1**A** (1301µs); client, 1**P** + 1**F** + 1**A** (235µs).
3. QUIC: server, 2**A** (100µs); client, 1**F** + 2**A** + 1**P** (184µs).

(The operations needed for the client to verify the certificate chain are not included.)

If we pick common primitives for each of these (RSA 2048 for the public and private operations, ECDH P-256 for TLS forward security and Curve25519 for QUIC) then we obtain the example times in parenthesis on an i7-3770S. If QUIC were to use P-256 then the server times would be 300µs and the client would be 385µs, so a fair amount of the gain comes from using better primitives.

TLS session resumption rates in the wild are roughly 50%, but QUIC doesn't include session resumption explicitly. However, it can achieve many of the benefits of resumption without any support in the protocol by having clients and servers maintain a cache of Diffie-Hellman results. This optional cache eliminates the computational burden of handshaking several times with the same server so long the client hasn't rotated its ephemeral key. If we assume a 50% resumption rate for TLS and assume that the QUIC cache does nothing, then we get the roughly 5x speedup in relation to TLS that was mentioned in the introduction.

Strike-register design

The strike-register, conceptually, implements a single function: Insert(current_time, nonce), which returns true if it can establish that the given nonce has never been inserted before.

The nonce contains a timestamp from the client, the server's orbit and 20 random bytes. The strike-register attempts to maintain a window around the current time in which it is authoritative.

If we assume that a strike-register takes 32 bytes to store each entry (which our design does) then, at 1000 connections per second, 1GiB of memory is sufficient to store more than 9 hours of nonces.

Having the strike-register establish a large window of authority allows the server to tolerate large client clock skews. But there is a motivation to keep the window of validity small if using an in-memory data structure: when restarting, the strike-register must reject nonces that it could previously have accepted. If the strike-register was previously accepting nonces from 4 hours into the future then, when restarting, it must reject nonces with timestamps in the next four hours in case it previously accepted them but dropped the data when restarting.

The strike-register must also be resilient to algorithmic complexity attacks since it will be processing arbitrary input from the outside world. Our design uses a crit-bit tree, which maintains acceptable performance even for pathological inputs. The crit-bit tree also allows the efficient removal of the entry with the oldest timestamp when the register is full. When removing an entry, the "horizon" is updated to the timestamp of that entry and all nonces with a timestamp equal to, or before that horizon will now be rejected.

This leads to a possible attack: by flooding the server with nonces at the limit of the strike-register's window, the attacker may be able to push the horizon past the current time. This would lead to all clients with a good clock to take an extra round trip in order to connect because their nonces would be rejected.

One effective means to counter this is to overprovision the strike-register. If the time window is 10 minutes, then it takes very little memory for the strike-register to be able to handle much more traffic than could be handled by the rest of the system.

Alternatively, the "far horizon" (the future time past which the strike-register will reject nonces) can be set to be the same distance from the current time as the horizon is. Typically the far horizon is the current time plus the maximum window. However, if the horizon starts to move in, because the strike-register is full, then the far horizon can be moved in by the same amount. Nonces that are now past the far horizon have to continue to be stored, but the attack now only manages to reduce the amount of clock skew tolerated.

There are two cases where the strike-register may present persistent problems (i.e. a server may not be able to terminate connections.) Firstly, the client's clock-skew may be greater than the tolerance of the strike-register leading to repeated connection rejections. Secondly, the strike-register may fail: either because it has just been started and is in its quiescent period, or because it's a shared service for a number of servers and that service has failed.
The first of these issues could be addressed by having the clients take a clock-sync signal from the server and define a "server local time" offset for that server. But the second problem cannot be so easily solved and so we solve both of them with optional server nonces.

If a 0-RTT handshake attempt fails, the server will provide a server nonce which the client can echo in the next handshake attempt. The server nonce is an encrypted and authenticated timestamp + nonce (the details of the encryption are purely a server-internal matter). The client can echo a server nonce and, if it's needed to establish uniqueness for the connection, the server will decrypt it and insert the nonce into a separate, local strike-register. Because the strike-register is local, and the timestamp came from the server, it will never fail unless full. Thus client clock-sync issues and shared strike-register failures don't prevent the handshake from completing (although they will cost a round trip).

When the local strike-register fills up, the horizon will advance and earlier server nonces will

become invalid. However, the server nonce strike-register needs only to be able to store enough entries to handle the connection load for a few round trips and, at 32 bytes per entry, that shouldn't be a problem.

Because the server encrypted and authenticated the nonce itself, with a randomly generated key, it knows that the nonce value came from the same instance of itself originally, and therefore that the server nonce strike-register is always authoritative. Also, since the key is generated at server startup, no startup quiescence period is needed to ensure that old values aren't accepted twice after losing state.

The downside to this is that it assumes that subsequent handshake attempts will be handled by the same server so that the server nonce is acceptable. The load balancer design has to ensure this.

Wire Protocol

QUIC is a datagram protocol, and the full payload of each datagram (above the UDP layer) are authenticated and encrypted once keys have been established. The underlying datagram protocol provides the crypto layer with the means to send reliable, arbitrary sized messages. These messages have a uniform, key-value format.

The keys are 32-bit *tags*. This seeks to provide a balance between the tyranny of magic number registries and the verbosity of strings. As far as the wire protocol is concerned, these are opaque, 32-bit values and, in this document, tags will often be written like `EXMP`. Although it's written as a string, it's just a mnemonic for the value 0x504d5845. That value, in little-endian, is the ASCII string E X M P.

If a tag is written in ASCII but is less than four characters then it's as if the remaining characters were NUL. So `EXP` corresponds to 0x505845.

If the tag value contains bytes outside of the ASCII range, they'll be written in hex, e.g. 504d5845.

All values are little-endian unless otherwise noted.

A handshake message consists of:

1. The tag of the message.
2. A uint16 containing the number of tag-value pairs.
3. Two bytes of padding which should be zero when sent but ignored when received.
4. A series of uint32 tags and uint32 end offsets, one for each tag-value pair. The tags must be strictly monotonically increasing, and the end-offsets must be monotonic non-decreasing. The end offset gives the offset, from the start of the value data, to a byte one beyond the end of the data for that tag. (Thus the end offset of the last tag contains the length of the value data).

5. The value data, concatenated without padding.

The tag-value format allows for an efficient binary search for a tag after only a small fraction of the data has been validated. The requirement that the tags be strictly monotonic also removes any ambiguity around duplicated tags.

Although the 32-bit lengths are currently more than needed, 16-bit lengths ran the risk of being insufficient to handle larger, post-quantum values.

Any message may contain a padding (PAD) tag. These can be used to to defeat traffic analysis. Additionally, we may define a global minimum size for client hellos to limit amplification attacks. Client hellos that are smaller than the minimum would need a PAD tag to make up the difference.

Client handshake.

The flow of a client handshake is illustrated in figure 1. Conceptually, all handshakes in QUIC are 0-RTT, it's just that some of them fail and need to be retried.

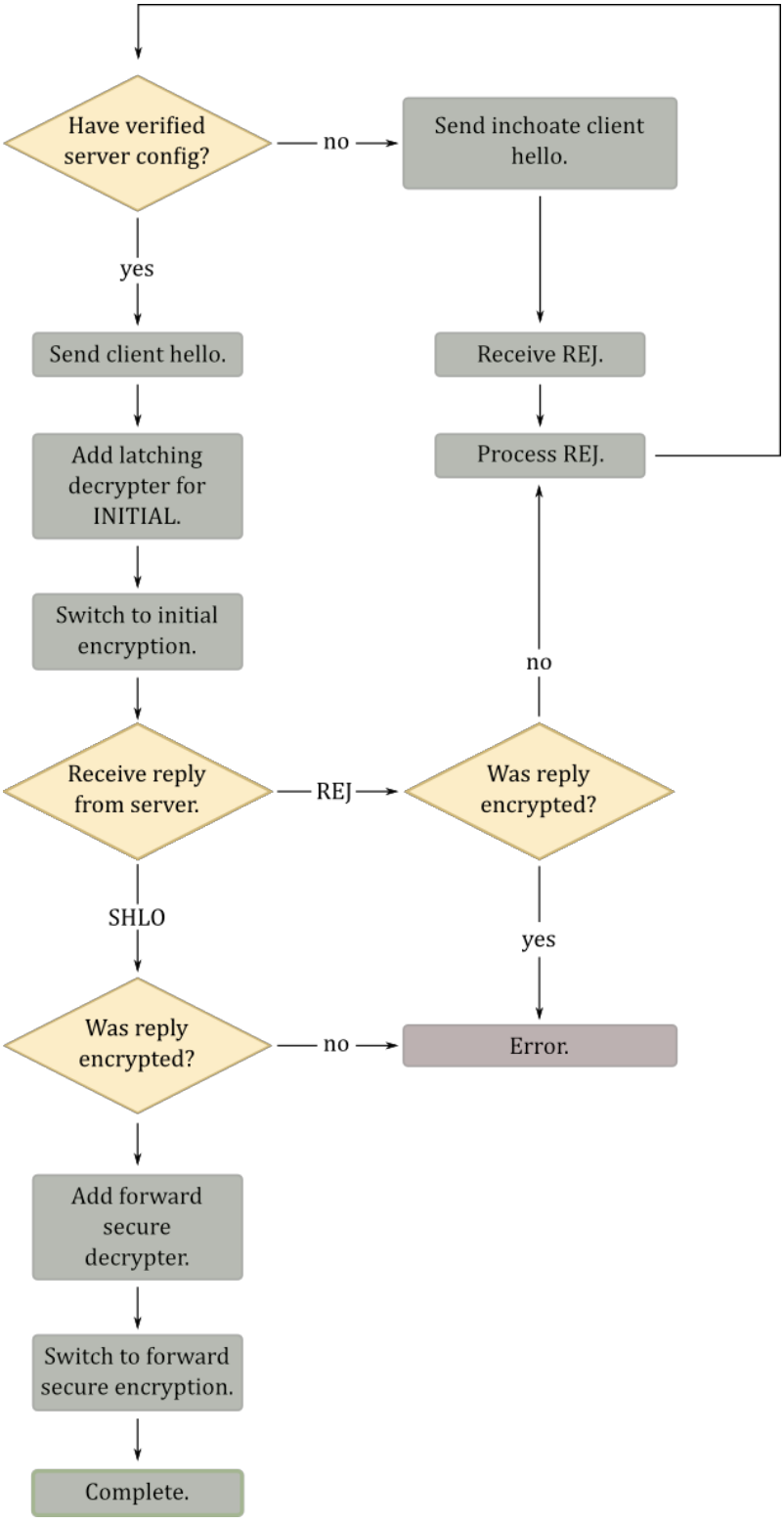

Figure 1. Client handshake flow.

In order to perform a 0-RTT handshake, the client needs to have a server config that has been

verified to be authentic. Initially we assume that the client knows nothing about the server and so, before a handshake can be attempted, the client will send "inchoate" client hello messages to elicit a server config and proof of authenticity from the server. There may be several rounds of inchoate client hellos before the client receives all the information that it needs because the server may be unwilling to send a large proof of authenticity to an unvalidated IP address.

Client hello messages have the message tag `CHLO` and, in their inchoate form, contain the following tag/value pairs:

`SNI`	Server Name Indication (optional): the fully qualified DNS name of the server canonicalised to lowercase with no trailing period. Internationalized domain names nee to be encoded as A-labels defined in RFC 5890. The value of the `SNI` tag must not be an I address literal.
`STK`	Source-address token (optional): the source-address token that the server has previousl provided, if any.
`PDMD`	Proof demand (optional): a list of tags describing the types of proof acceptable to the client, in preference order. Currently only `X509` is defined.
`CCS`	Common certificate sets (optional): a series of 64-bit, FNV-1a hashes of sets of commor certificates that the client possesses. (See section on certificate compression.)
`CCRT`	Cached certificates (optional): a series of 64-bit, FNV-1a hashes of cached certificates for this server. (See section on certificate compression.)
`VERS`	Version: a single tag that mirrors the protocol version advertised by the client in eacl QUIC packet.

(Other parts of QUIC may define additional tags to be included in the client and server hellos. For example, the maximum number of stream, congestion control parameters etc. However, those tags are not defined in this specification.)

In response to a client hello the server will either send a rejection message, or a server hello. The server hello indicates a successful handshake and can never result from an inchoate client hello as it doesn't contain enough information to perform a handshake. The rejection messages contain information that the client can use to perform a better handshake attempt subsequently.

Rejection messages have the tag `REJ` and contain the following tag/value pairs:

`SCFG`	Server config (optional): a message containing the server's serialised config. (Describe below.)
`STK`	Source-address token (optional): an opaque byte string that the client should echo ir

future client hello messages.

SNO Server nonce (optional): the server may set an nonce, which the client should echo in any future (full) client hello messages. This allows a server to operate without a strike-register and for clients with clock-skew to connect.

ff54 5243 Certificate chain (optional): the server's certificate chain. (See section on certificate compression.)

PROF Proof of authenticity (optional): in the case of X.509, a signature of the server config by the public key in the leaf certificate. The format of the signature is currently fixed by the type of public key:

RSA	RSA-PSS-SHA256
ECDSA	ECDSA-SHA256

The signature is calculated over:

1. The label "QUIC server config signature"
2. An 0x00 byte
3. The serialised server config.

Although all the elements of the rejection message are optional, the server must allow the client to make progress. For example, if the client didn't present a source-address token and the server is unwilling to send the server config to an unverified IP address, then the server must reply with a source-address token so that the client's next handshake attempt will be more successful.

Some tags are specified in hex rather than in the ASCII notation. This is because the tags are formed so that they will either come at the beginning or end of a message. Recall that tags, as numbers, are written in little-endian order on the wire.

Tags containing entropy are moved to the beginning of the message as the server may not be maintaining state and therefore may process a duplicated client hello twice. If packet loss hits the reject message and the entropy fields spanned a packet boundary then the client may misassemble them.

Large tags (the certificate chain so far) are moved to the end of the message so that they don't delay the receipt of other fields that may be sufficient.

The server config contains the serialised preferences for the server and takes the form of a handshake message with tag SCFG. It contains the following tag/value pairs:

SCID Server config ID: an opaque, 16-byte identifier for this server config.

KEXS Key exchange algorithms: a list of tags, in preference order, specifying the key exchang

algorithms that the server supports. The following tags are defined:

C255	Curve25519
P256	P-256

AEAD Authenticated encryption algorithms: a list of tags, in preference order, specifying th AEAD primitives supported by the server. The following tags are defined:

AESG	AES-GCM with a 12-byte tag and IV. The first four bytes of the IV are taken from the key derivation and the last eight are the packet sequence number.
S20P	Salsa20 with Poly1305. (Provisional and not yet implemented.)

PUBS A list of public values, 24-bit, little-endian length prefixed, in the same order as in KEXS.

ORBT Orbit: an 8-byte, opaque value that identifies the strike-register.

EXPY Expiry: a 64-bit expiry time for the server config in UNIX epoch seconds.

VERS Versions: the list of version tags supported by the server. The underlying QUIC packe protocol has a version negotiation. The server's supported versions are mirrored in th signed server config to confirm that no downgrade attack occurred.

Once the client has received a server config, and has authenticated it by verifying the certificate chain and signature, it can perform a handshake that isn't designed to fail by sending a full client hello. A full client hello contains the same tags as an inchoate client hello, with the addition of several others:

SCID Server config ID: the ID of the server config that the client is using.

AEAD Authenticated encryption: the tag of the AEAD algorithm to be used.

KEXS Key exchange: the tag of the key exchange algorithm to be used.

NONC Client nonce: 32 bytes consisting of 4 bytes of timestamp (big-endian, UNIX epoch seconds), 8 bytes of server orbit and 20 bytes of random data.

SNO Server nonce (optional): an echoed server nonce, if the server has provided one.

PUBS Public value: the client's public value for the given key exchange algorithm.

CETV Client encrypted tag-values (optional): a serialised message, encrypted with the AEAI

algorithm specified in this client hello and with keys derived in the manner specified i the CETV section, below. This message will contain further, encrypted tag-value pairs tha specify client certificates, ChannelIDs etc.

After sending a full client hello, the client is in possession of non-forward-secure keys for the connection since it can calculate the shared value from the server config and the public value in PUBS. (For details of the key derivation, see below.) These keys are called the initial keys (as opposed the the forward-secure keys that come later) and the client should encrypt future packets with these keys. It should also configure the packet processing to accept packets encrypted with these keys in a latching fashion: once an encrypted packet has been received, no further unencrypted packets should be accepted.

At this point, the client is free to start sending application data to the server. Indeed, if it wishes to achieve 0-RTT then it must start sending before waiting for the server's reply.

Retransmission of data occurs at a layer below the handshake layer, however that layer must still be aware of the change of encryption. New packets must be transmitted using the initial keys but, if the client hello needs to be retransmitted, then it must be retransmitted in the clear. The packet sending layer must be aware of which security level was originally used to send any given packet and be careful not to use a higher security level unless the peer has acknowledged possession of those keys (i.e. by sending a packet using that security level).

The server will either accept or reject the handshake. In the event that the server rejects the client hello, it will send a REJ message and all packets transmitted using the initial keys must be considered lost and need to be retransmitted under the new, initial keys. Because of this, clients should limit the amount of data outstanding while a server hello or rejection is pending.

In the happy event that the handshake is successful, the server will return a server hello message. This message has the tag SHLO, is encrypted using the initial keys, and contains the following tag/value pairs in addition to those defined for a rejection message:

PUBS An ephemeral public value for the key exchange algorithm used by the client.

With the ephemeral public value in hand, both sides can calculate the forward-secure keys. (See section on key derivation.) The server can switch to sending packets encrypted with the forward-secure keys immediately. The client has to wait for receipt of the server hello. (Note: we are considering having the server wait until it has received a forward-secure packet before sending any itself. This avoids a stall if the server hello packet is dropped.)

Key derivation

Key material is generated by an HMAC-based key derivation function (HKDF) with hash function SHA-256. HKDF (specified in RFC 5869) uses the approved two-step key derivation procedure specified in NIST SP 800-56C.

Step 1: HKDF-Extract
The output of the key agreement (32 bytes in the case of Curve25519 and P-256) is the premaster secret, which is the input keying material (*IKM*) for the HKDF-Extract function. The *salt* input is the client nonce followed by the server nonce (if any). HKDF-Extract outputs a pseudorandom key (*PRK*), which is the master secret. The master secret is 32 bytes long if SHA-256 is used.

Step 2: HKDF-Expand
The *PRK* input is the master secret. The *info* input (context and application specific information) is the concatenation of the following data:

1. The label "QUIC key expansion"
2. An 0x00 byte
3. The GUID of the connection from the packet layer.
4. The client hello message
5. The server config message

Key material is assigned in this order:

1. Client write key.
2. Server write key.
3. Client write IV.
4. Server write IV.

If any primitive requires less than a whole number of bytes of key material, then the remainder of the last byte is discarded.

When the forward-secret keys are derived, the same inputs are used except that *info* uses the label "QUIC forward secure key expansion".

Client encrypted tag values

A client hello may contain a CETV tag in order to specify client certificates, ChannelIDs and other non-public data in the client hello. (That's in contrast to TLS, which sends client certificates in the clear.)

The CETV message is serialised and encrypted with the AEAD that is specified in the client hello. The key is derived in the same way as the keys for the connection (see Key Derivation, above) except that *info* uses the label "QUIC CETV block". The client hello message used in the derivation is the client hello without the CETV tag. When the connection keys are later derived, the client hello used will include the CETV tag.

The AEAD nonce is always zero, which is safe because only a single message is ever encrypted with the key.

Proving possession of a private key, which is needed for both client certificates and ChannelIDs, is done by signing the HKDF *info* input that is used in the CETV key derivation.

The CETV message can include the following tags:

`CIDK`	ChannelID key (optional): a pair of 32-byte, big-endian numbers which, together, specify an (x, y) pair. This is a point on the P-256 curve and an ECDSA public key.
`CIDS`	ChannelID signature (optional): a pair of 32-byte, big-endian numbers which, together, specify the (r, s) pair of an ECDSA signature of the HKDF input.

Certificate compression

In TLS, certificate chains are transmitted uncompressed and take up the vast majority of the bytes in full handshakes. In QUIC, we hope to be able to avoid some round trips by compressing the certificates.

A certificate chain is a series of certificates which, for the purposes of this section, are opaque byte strings. The leaf certificate is always first in the chain and the root CA certificate should never be included.

When serialising a certificate chain in the `CERT` tag of a rejection message, the server considers what information the client already has. This prior knowledge can come in two forms: possession of bundles of common intermediate certificates, or cached certificates from prior interactions with the same server.

The former are expressed as a series of 64-bit, FNV-1a hashes in the `CCS` tag of the client hello. If both the client and server share at least one common certificate set then certificates that exist in them can simply be referenced.

The cached certificates are expressed as 64-bit, FNV-1a hashes in the `CCRT` tag of the client hello. If any are still in the certificate chain then they can be replaced by the hash.

Any remaining certificates are gzip compressed with a pre-shared dictionary that consists of the certificates specified by either of the first two methods, and a block of common strings from certificates taken from the Alexa top 5000.

The concrete representation of this is placed into the `CERT` tag of the rejection message and has the format of a `Cert` structure in the following TLS presentation style:

```
enum { end_of_list(0), compressed(1), cached(2), common(3) } EntryType;

struct {
  EntryType type;
  select (type) {
    case compressed:
      // nothing
    case cached:
      opaque hash[8];
    case common:
      opaque set_hash[8];
      uint32 index;
  }
} Entry;

struct {
  Entry entries[];
  uint32 uncompressed_length;
  opaque gzip_data[];
} Certs;
```

(Recall that numbers are little-endian in QUIC.)

The `entries` list is terminated by an `Entry` of type `end_of_list`, rather than length-prefixed as would be common in TLS. The `gzip_data` extends to the end of the value.

The gzip, pre-shared dictionary contains the certificates of type `compressed` or `cached`, concatenated in reverse order, followed by ~1500 bytes of common substrings that are not given here.

Future directions

1. It's likely that ChannelID will be removed from this layer of the protocol and, instead, the crypto handshake will produce a channel binding value that can be signed at a higher layer.
2. Trevor Perrin has pointed out that the server can return a server can return an encrypted ticket containing Hash(forward secure secret) that the client could echo to the server on future connections. This would save one Diffie-Hellman operation for those handshakes.

3. Servers should be able to indicate to clients that they should wait until forward secure keys are established before sending application data.
4. In order to avoid head-of-line blocking by the server hello packet, the server could avoid sending forward secure data until the client confirms receipt of server hello. (For example: by sending a forward secure packet itself.)

Acknowledgements

Thanks to Trevor Perrin, Ben Laurie and Emilia Käsper for their valuable feedback.