Token Binding over QUIC

Chromium intent-to-implement:
https://docs.google.com/document/d/1Ta3GlT_LrqAOLV217Kutn3B2trvifStxB0CThQ_kk78/edit
Specs:
http://datatracker.ietf.org/doc/draft-ietf-tokbind-protocol/
http://datatracker.ietf.org/doc/draft-ietf-tokbind-negotiation/
http://datatracker.ietf.org/doc/draft-ietf-tokbind-https/

The current Token Binding draft specs are designed around an application protocol (such as HTTP) running over TLS. In addition to supporting Token Binding over TLS, we wish to support Token Binding using QUIC in place of TLS. This is broken into two parts: the first is how to negotiate Token Binding with QUIC (replacing “Transport Layer Security (TLS) Extension for Token Binding Protocol Negotiation”), and the second is making a slight change to the Token Binding Protocol to support QUIC in place of TLS.

Changes to QUIC crypto to support Token Binding

QUIC_VERSION_26 introduces a new tag, XLCT, which must be present in the client hello, and makes a change to the main KDF, so that it closes over the server’s identity. These changes are to avoid an unknown key share attack on Token Binding.

The value of the XLCT tag sent by the client in the CHLO must be equal to the FNV-1a hash of the server’s leaf certificate. (The XLCT name is an abbreviation for “expected leaf certificate.”) The server must verify this value, and send a REJ message if the values do not match. Such a mismatch could occur in a 0-RTT connection where the server has rotated its certificate since the client last connected.

The main KDF is changed so that the HKDF info input includes the server’s leaf certificate. The info input previously had a suffix containing the connection ID, serialized client hello, and serialized server config. This change appends to the end of that suffix the server’s leaf certificate.

Token Binding Protocol Negotiation in QUIC

A new tag, TBKP, in the server config and the full client hello will be used to negotiate the token binding key parameters. In the server config, the value for this tag will be a list of tags corresponding to the key parameters that the server supports. Currently only one key parameter is supported, ecdsap256, represented by tag P256, corresponding to the value in the TokenBindingKeyParameters enum from the Token Binding Negotiation TLS Extension spec. If the client chooses to negotiate Token Binding, the full client hello will include the tag TBKP with a value of the tag for the preferred key parameter in the server’s list, or omit the TBKP tag if the

client does not support any of the server's key parameters. This should function the same way as AEAD or KEXS.

This negotiation scheme also works for 0-RTT connections: The client already knows what the server supports (because it is in the server config), so it sends a single value for TBKP in the client hello, and in the application data it can send a TokenBindingMessage (e.g. a Token-Binding header) using a key of that negotiated type.

Token Binding Protocol changes

The Token Binding Protocol assumes that TLS will be used, but the only part of the spec that is specific to using TLS for the transport layer is the reference to its Keying Material Exporters (as defined in RFC 5705).

Key Material Exporters

Quic has its own keying material exporter (as defined in QuicCryptoStream::ExportKeyingMaterial) which appears to be a likely candidate to use in place of the TLS EKM. However, it cannot be used until the crypto handshake is complete because it uses a subkey secret that is generated as part of the forward secure key derivation. To work around this issue, we can generate a subkey secret during the initial key derivation, and use that in a new exporter.

The following changes would be needed:

- Add a initial_subkey_secret field to QuicCryptoNegotiatedParameters
- In the client and server, when calling CryptoUtils::DeriveKeys to derive the initial keys, pass in &initial_subkey_secret to be filled.
- Define a new method QuicCryptoStream::ExportInitialKeyingMaterial. This will be implemented using CryptoUtils::ExportKeyingMaterial (like QuicCryptoStream::ExportKeyingMaterial), but uses initial_subkey_secret instead of subkey_secret.