Handshake workflow in TLS 1.2 vs 1.3

Ganyu (Bruce) Xu

August 12, 2024

1 Handshake workflow in TLS 1.2 (RFC 5246)

TLS 1.2 supports three main key exchange workflows: long-term Diffie-Hellman (DH and ECDH), RSA, and ephemeral Diffie-Hellman (DHE and ECDHE). Depending on the choice of key exchange method, the handshake workflow may vary, but there are three invariants:

- Some key exchange so client and server can establish shared secret over insecure channel
- Server needs to authenticate itself by presenting some kind of public key signed by a certificate authority, then prove posession of the corresponding secret key
- Some kind of key confirmation to confirm that client and server indeed have the same secret

1.1 With long-term Diffie-Hellman

1. **Client sends** `ClientHello`:
   in `ClientHello`, client specifies that it supports `DH_*` for key exchange
2. **Server responds with** `ServerHello`
   in `ServerHello`, server chooses `DH_*` for key exchange
3. **Server sends** `Certificate`
   `Certificate` contains server's Diffie-Hellman parameters g^y signed by the certificate authority (RFC 5246 A.5). Also note that with a `DH_*` key exchange, *server must not send* ***`ServerKeyExchange`***, or the client will abort the handshake (7.4.3)
4. **Client sends** `ClientKeyExchange`
   `ClientKeyExchange` contains client's Diffie-Hellman parameters.
   After server receives `ClientKeyExchange`, client and server have agreed on a pre-master secret, and the handshake traffic keys are derived from pre-master secret.
5. **Client and server send each other** `Finished`
   `Finished` contains a MAC over all previous messages signed using a secret key derived from the pre-master secret.

With long-term Diffie-Hellman, **server authentication is implicit**. Server demonstrates possession of the Diffie-Hellman secret key that correspond to the public key in `Certificate` by correctly computing the shared secret, then use the shared secret to compute a valid tag, which is presented in `Finished`. If the server does not have the corresponding secret key, then it cannot produce a valid tag, so the client will abort the handshake after receiving an invalid `Finished` from the server.

1.2 With RSA

1. **Client sends** `ClientHello`:
   in `ClientHello`, client specifies that it supports `RSA_*` for key exchange

2. **Server responds with** `ServerHello`
   in `ServerHello`, server chooses `RSA_*` for key exchange

3. **Server sends** `Certificate`
   `Certificate` contains server's RSA public key (an encryption key) signed by the certificate authority (RFC 5246 A.5). Similar to long-term Diffie-Hellman key exchange, *server must not send `ServerKeyExchange`*.

4. **Client sends** `ClientKeyExchange`
   `ClientKeyExchange` contains the RSA ciphertext generated by the client using the server's public key in the certificate.
   After server receives `ClientKeyExchange`, client and server have agreed on a pre-master secret.

5. **Client and server send each other** `Finished`
   `Finished` contains a MAC over all previous messages signed using a secret key derived from the pre-master secret.

In RSA key exchange, **server authentication** is also implicit just like with long-term Diffie-Hellman key exchange.

1.3 With ephemeral Diffie-Hellman

1. **Client sends** `ClientHello`:
   in `ClientHello`, client specifies that it supports `DHE_*` for key exchange

2. **Server responds with** `ServerHello`
   in `ServerHello`, server chooses `DHE_*` for key exchange

3. **Server sends** `Certificate`
   `Certificate` contains server's public key (a digital signature signing key) signed by the certificate authority

4. **Server sends** `ServerKeyExchange`
   `ServerKeyExchange` contains server's ephemeral Diffie-Hellman parameters and a digital signature of these server parameters signed using server's secret key corresponding to the public key in server certificate

5. **Client sends** `ClientKeyExchange`
   contains client's ephemeral Diffie-Hellman parameters

6. **CLient and server send each other** `Finished`

In DHE, **server authentication is explicit**: server presents its public key (a verification key) in the server certificate, which is signed by the the certificate authority and thus bound to the server's identity. Server sends demonstrates possession of the corresponding secret key (a signing key) by signing client secret, server secret, and server DHE parameters in `ServerKeyExchange`.

2 Key exchange workflow in TLS 1.3

TLS 1.3 differs from 1.2 in that **TLS 1.3 deprecated all long-term Diffie-Hellman and RSA key exchange methods; ephemeral Diffie-Hellman key exchange is the only supported key exchange**. The handshake workflow thus differs from TLS 1.2.

1. **Client sends** `ClientHello`
    - The `cipher_suites` field contains the supported set of cipher suites, but *this field no longer specifies the key exchange methods*, only the AEAD scheme (AES-128-GCM, AES-256-GCM or ChaCha20-Poly1305) and HKDF (SHA256 or SHA384)
    - The `random` field contains client's random values
    - The `signature_algorithms` extension specifies the type of signature the client supports for `CertificateVerify` (aka server's signature).
    - The `signature_algorithms_sert` extension specifies the type of signature the client supports for `Certificate` (aka certificate authority's signature)
    - The `supported_group` extension specifies the Diffie-Hellman groups that the client supports
    - The `key_share` extension specifies client's ephemeral Diffie-Hellman parameters (g^x)
2. **Server responds with** `ServerHello`
    - The `random` field contains server's random values
    - The `key_share` extension contains server's chosen Diffie-Hellman group and the corresponding ephemeral parameters
3. **Server sends** `Certificate`
   In TLS 1.3, *server's certificate must contain a digital signature's verification key.* This public key is bound to the server's identity by certificate authority's signature
4. **Server sends** `CertificateVerify`
   This message contains a signature over all handshake messages up to this point, signed using the server's secret key. **This authenticates the server explicitly**.
5. Client authentication is optional in TLS. If client authentication is needed, then server requests client's certificate with `CertificateRequest`, and client respond with client's authentication messages `Certificate` and `CertificateVerify`.
6. Client and server send each other `Finished`, which contains a MAC over the handshake transcript signed using the shared secret

3 Incorporating post-quantum KEM into TLS 1.3

While there is no standardization on how post-quantum KEMs will be incorporated into TLS 1.3, Google and Cloudflare have done some experimental implementations in 2019. At the time the chosen KEMs were `HRSSS` and `SIKE`, though there is nothing special about these two KEMs, so any other KEM can be a drop-in replacement. The handshake workflow will be as follows:

1. `ClientHello`
    - In the `supported_groups` extensions, add a group that indicates client's support for post-quantum (hybrid) key exchange, such as `Kyber768X25519`
    - In the `key_share` extension, include a key share entry whose value is the concatenation of classical and PQ public keys `key_share = X25519_public_value || ephemeral_KEM_public_key`
2. `ServerHello`
    - In the `key_share` extension, server's response is the concatenation of server's ephemeral Diffie-Hellman parameters and the ciphertext produced by running the encapsulation routine using client's ephemeral encapsulation key

The pre-master secret is derived from the concatenation of the Diffie-Hellman shared secret and the KEM shared secret.

In this setup, a fresh KEM key pair is generated by the client at each handshake, which means that each decapsulation key is used exactly once. If the decapsulation routine rejects the ciphertext explicitly, or if the client and the server's pre-master secret disagree, then somewhere along the handshake protocol the client will abort the handshake. When the handshake is restarted, another key pair will be generated. Therefore, at least in the context of TLS, **limiting a CCA adversary to exactly one decryption query** is a reasonable security goal.