TLS 1.3 Status

Eric Rescorla

Mozilla

ekr@rtfm.com

Overview of Changes Since IETF 92 (Major)

- Integrate DH-based handshake (per WG discussion in Dallas)
- Add initial cut at 0-RTT support
- HKDF-based key derivation (per WG discussion in Dallas)
- Moved ClientKeyShare into an extension
- Added support for PSK
- Removed resumption and merged ticket support with PSK

Overview of Changes Since IETF 92 (Minor)

- Prohibit RC4 negotiation
- Froze record-layer header
- Context field for signatures
- Replaced explicit IV with sequence number + mask

Open Issues Preview

- Indicating known configurations
- 0-RTT w/ PSK
- Interaction of 0-RTT and authentication
- 0-RTT rejection handling
- PSK resumption restrictions
- Traffic key generation

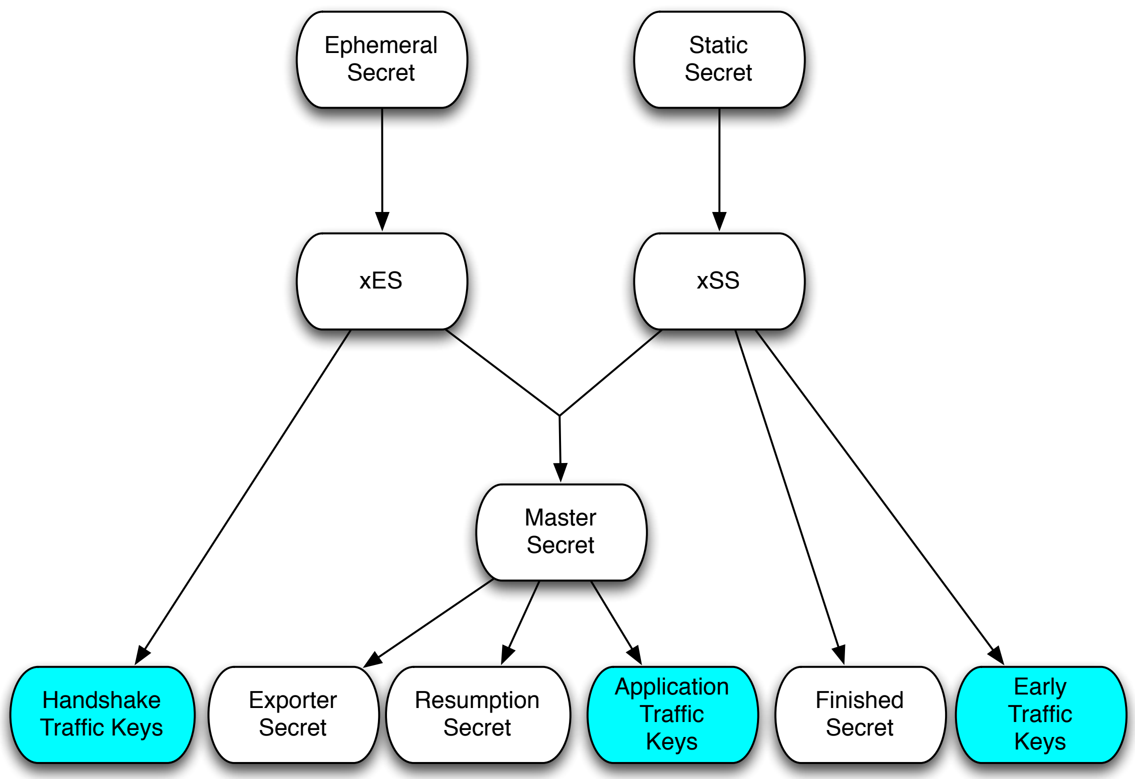
DH-Based Handshake (Review)

- Server has a semi-static DH key (just like 1-RTT)
- Probably really has long-term signing key
  - Used to sign the semi-static key
  - Agreement at previous IETFs to use online-only signing
- Common key exchange computations between all modes

Key Computation Inputs

Key Exchange	Static Secret (SS)	Ephemeral Secret (ES)
(EC)DHE (full handshake)	Client ephemeral w/ server ephemeral	Client ephemeral w/ server ephemeral
(EC)DHE (w/ known_configuration)	Client ephemeral w/ Known Key	Client ephemeral w/ server ephemeral
PSK	Pre-Shared Key	Pre-shared key
PSK + (EC)DHE	Pre-Shared Key	Client ephemeral

Key Computations



Two New Mechanisms

- Server configurations and known configuration
  - Server publishes a configuration to the client in handshake n
  - Client reuses that configuration in handshake $n + 1$
- Early data indication
  - Client indicates that he wants to do 0-RTT (client auth, data, both)
  - Server accepts or rejects

Example: Initial Handshake

```
ClientHello
  + ClientKeyShare        -------->
                                                ServerHello
                                            ServerKeyShare*
                                      {EncryptedExtensions}
                                    {ServerConfiguration*} <- SEE HERE
                                             {Certificate*}
                                      {CertificateRequest*}
                                       {CertificateVerify*}
                          <--------              {Finished}
{Certificate*}
{CertificateVerify*}
{Finished}                -------->
[Application Data]        <------->      [Application Data]
```

Known Configuration

```
struct {
    opaque configuration_id<1..2^16-1>;
    uint32 expiration_date;
    NamedGroup group;
    opaque server_key<1..2^16-1>;
    Boolean early_data_allowed;
} ServerConfiguration;
```

- The client's reuse of the configuration implicitly resurrects the previous state (See open issues)

Example: 0-RTT Handshake (w/o new configuration)

```
ClientHello
  + ClientKeyShare
  + KnownConfiguration
  + EarlyDataIndication
(Certificate*)
(CertificateVerify*)
(Application Data)        -------->
                                                ServerHello
                                       + KnownConfiguration
                                      + EarlyDataIndication
                                             ServerKeyShare
                          <--------              {Finished}
{Finished}                -------->

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

Early Data Indication

```
enum { early_handshake(1), early_data(2),
       early_handshake_and_data(3), (255) } EarlyDataType;

struct {
  select (Role) {
    case client:
      opaque context<0..255>;
      EarlyDataType type;
    case server:
      struct {};
  }
} EarlyDataIndication;
```

What do failed 0-RTT handshakes look like?

- Server doesn't respond with an EarlyDataIndication
  - System falls back to 1-RTT
  - All of the early data is just ignored
- This is kind of clunky
  - Early handshake messages have a different content type
  - What about encrypted content types
- Analysis needed that ignoring early data is OK
  - ... currently underway

Managing semi-static keys (I)

- Need two keys
  - Ephemeral (for PFS)
  - Semi-static (cached server 1-RTT, 0-RTT)
- Various options for making these work together
  - Always use a single semi-static key – suboptimal performance
  - Have the server supply a separate key – odd when you refresh keys

Managing semi-static keys (II)

- Current draft state
  - First handshake looks like draft-06
    - Can supply a `ServerConfiguration`
  - Subsequent handshakes can reuse `ServerConfiguration`
    - But need to sign if they want to provide one
- More modes than we would really like (but best perf profile)

Example: 0-RTT Handshake w/ new configuration

```
ClientHello
  + ClientKeyShare
  + KnownConfiguration
  + EarlyDataIndication
(Certificate*)
(CertificateVerify*)
(Application Data)        -------->
                                                   ServerHello
                                          + KnownConfiguration
                                         + EarlyDataIndication
                                                ServerKeyShare
                                     {ServerConfiguration*} <- SEE HERE
                                             {Certificate*} <- SEE HERE
                                       {CertificateVerify*} <- SEE HERE
                          <--------                 {Finished}
{Finished}                -------->

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

Pre-Shared Keys

- TLS 1.2 had PSK
  - But we kind of broke it
- draft-07 brings it back
  - But I did get rid of identity hint...

Example: Pure PSK Handshake

```
ClientHello
  + ClientKeyShare,
    PreSharedKeyExtension -------->
                                                  ServerHello
                                       +PreSharedKeyExtension
                          <--------                 {Finished}
{Certificate*}
{Finished}                -------->
[Application Data]        <------->         [Application Data]
```

- Can also do this with DHE-PSK

PreSharedKey Extension

```
opaque psk_identity<0..2^16-1>;

struct {
  select (Role) {
    case client:
      psk_identity identities<0..2^16-1>;

    case server:
      psk_identity identity;
} PreSharedKeyExtension;
```

PSK For Resumption

- Resumption and PSK are very similar
  - Let's make them identical
- Basic idea
  - Server gives client a PSK label
  - PSK is derived from initial handshake (resumption master secret)

Example: Establishing a PSK for resumption

```
ClientHello
  + ClientKeyShare        -------->
                                                ServerHello
                                             ServerKeyShare
                                      {EncryptedExtensions}
                                     {ServerConfiguration*}
                                             {Certificate*}
                                      {CertificateRequest*}
                                       {CertificateVerify*}
                          <--------              {Finished}
{Certificate*}
{CertificateVerify*}
{Finished}                -------->
                          <--------      [NewSessionTicket] <- SEE HER
[Application Data]        <------->      [Application Data]
```

ClientKeyShare Extension

- This used to be a separate message
  - That just made life complicated
- It's now an extension
- Nothing else has changed

Indicating Known Configurations

- Current design has client just indicate configuration ID
  - This means that the server needs to memorize each crypto configuration (ugh)
- Proposed redesign
  - Client indicates configuration ID *and* cryptographic configuration
    - Cipher suites and cryptographic extensions
    - MUST replicate the server's selection from a previous handshake
  - Server verifies client's `ClientHello`
    - Checks that configuration ID is valid
    - Verifies that client's parameters are what it would negotiate

Strawman

```
struct {
  select (Role) {
    case client:
      opaque identifier<0..2^16-1>;
      CipherSuite cipher_suite;             <- SEE HERE
      Extension extensions<0..2^16-1>;      <- SEE HERE

    case server:
      struct {};
  }
} KnownConfigurationExtension
```

Analysis

- Pros
  - Server doesn't need to keep per-connection state
  - Neatly solves PSK (and any other key negotiation mechanism)
  - Explicit state is explicit
- Cons
  - Server has to compare client's offer
  - Very modest wire bloat
- Note: we could have the server not echo the parameters in ServerHello
  - But I'd rather keep things consistent

0-RTT Rejection Handling (I)

- Currently it's all or nothing
  - Server can't accept 0-RTT client auth but not 0-RTT data
  - ... maybe it should be able to express its preferences in ServerConfiguration
- This seems easiest
- Proposed resolution: Server gets to indicate what it wants in ServerConfiguration

0-RTT Rejection Handling (II)

- How do you distinguish client's early data (which you want to discard) from the client's second flight (which you want to process)
- Current algorithm uses content type
  - Early handshake data has `early_handshake`
  - Early data has `application_data` type
  - The next thing you want to process has `handshake` type
  - Just skip to the next `handshake` message
- This isn't maximally elegant
  - And will fail with encrypted content types (there you need trial decryption)
  - Other ideas welcome

0-RTT Rejection Handling (III)

- What is included in handshake hash?
  - Handshake hash generally includes *plaintext*
  - ... but in rejection cases, you probably don't have decryption cases
- Present draft just ignores this data with rejection
- Alternative: include *ciphertext*
- Proposal: keep with current version pending analysis

0-RTT and Authentication

- There isn't any per-connection data from the server to sign
  - Client provides all the freshness*
- What context does the client have to sign?
  - It should include server identity

```
handshake_hash = Hash(
                      Hash(handshake_messages) ||
                      Hash(configuration)
                     )

configuration = ServerConfiguration || Certificate
```

*Insert caveats about issues with 0-RTT anti-replay

PSK Resumption Restrictions?

- Resumption required that you use the same ciphers
  - But if you make resumption PSK then you could in principle negotiate a new cipher
- Should we require servers to pick the same symmetric cipher?
- This would be somewhat easier if we had a la carte negotiation

AEAD IV

- TLS 1.2 (well, GCM) uses a partially explicit IV
  - This chews up bandwidth
- Consensus to remove explicit IV
  - And reuse sequence number
  - Brian Smith raised concerns about every connection using the same nonce sequence

draft-07 design for AEAD IV

- $iv\_length = max(8, N\_MAX)$
- Generate per-session mask of length iv_length
- Left-pad RSN with 0s to iv_length
- XOR RSN with mask to produce per-record nonce

Traffic Key Generation

- Presently we generate a `key_block`
- ... and then slice and dice
- Generating independent keys with a context input would be more HSM-friendly
- Expected context
  - Key length
  - Usage
  - Algorithm (ugh)
- Should we do this?

This slide intentionally left blank