TLS 1.3 Status
draft-10

Eric Rescorla
Mozilla

ekr@rtfm.com
Overview

• Changes since IETF 93 (Prague)
• Client authentication (PR#316)
• 0-RTT framing (#311, #295)
• HelloRetryRequest (Issues #104, #185)
• Re-key (#4, #125)
• Exporters (#282)
Changes Since IETF 93 (II)

- Always require digital signatures from the server with public-key cipher suites
  - ...even with 0-RTT
- Relaxed certificate selection rules *
- Deprecated a lot of algorithms *
- Encrypted content type *
- Built-in record padding *
- More context for key derivation *
- Improved CertificateRequest syntax *
Changes Since IETF 93 (II)

- Update key schedule
- Added MTI algorithms
- Reduced maximum record expansion
- Extensionsify ServerKeyShare
- AEAD now has no AAD
- Assorted editorial stuff
Relaxed Certificate Selection Rules

- TLS 1.2 requires that certificates appear in order
  - Many servers don’t do this
    * Not always possible
  - Many clients try to construct the path anyway
  - Updated draft to encourage but not require this

- TLS 1.2 required that server certificates conform to SignatureAlgorithms
  - But what if the only cert you have doesn’t match?
  - Draft now allows you to send it in that case
    * ...but only if you have to
Deprecated Algorithms

- Forbid MD5 (and SHA-224)
- Forbid SHA-1 in CertificateVerify
- Removed DSA
- Switched to PSS (more on this later)
- Removed a lot of old EC groups
Encrypted Content Type and Padding

```c
struct {
    ContentType opaque_type = application_data(23); /* see fragment.type */
    ProtocolVersion record_version = { 3, 1 }; /* TLS v1.x */
    uint16 length;
    aead-ciphered struct {
        opaque content[TLSPlaintext.length];
        ContentType type;
        uint8 zeros[length_of_padding];
    } fragment;
} TLSCiphertext;
```

- This allows padding
- But doesn’t require it
- Receiver behaves the same either way
Context for Key Derivation

struct HkdfLabel {
    uint16 length;
    opaque hash_value<0..255>:
    opaque label<9..255>;
};

• HSMs can look at the label value if they want

• Consensus was not to try to make something generic

• Presently traffic keys are one big block with slice-and-dice
  − I intend to split them up to make interfaces easier

• Objections?
Improved CertificateRequest Syntax (Popov)

struct {
    opaque certificate_extension_oid<1..2^8-1>;
    opaque certificate_extension_values<0..2^16-1>;
} CertificateExtension;

struct {
    SignatureAndHashAlgorithm
    supported_signature_algorithms<2..2^16-2>;
    DistinguishedName certificate_authorities<0..2^16-1>;
    CertificateExtension certificate_extensions<0..2^16-1>;
} CertificateRequest;

• Extensions correspond to X.509v3 extensions in the EE certificate
• Each extension has its own matching rule
  – KeyUsage and EKU defined in this document
• Client can ignore any unrecognized extensions
Client Authentication (PR#316)

- TLS 1.3 removed renegotiation
- But there’s still a need for servers to request certificates post-handshake
  - Especially in HTTP
- WG had consensus in Seattle to do something about this
- Formed ad hoc design team
  - AGL, DKG, EKR, Beurdouche, Bhargavan, Krawczyk, Langley, MT, Wee
Current Structure

ClientHello
  + ClientKeyShare --------->

        ServerHello
        ServerKeyShare*
        {EncryptedExtensions}
        {ServerConfiguration*}
        {Certificate*} <--\
        {CertificateRequest*} > Sign.
        {CertificateVerify*} <--/
        --------------
        {Finished} <- MAC

Sign.  /->  {Certificate*}
\->  {CertificateVerify*}

MAC  ->  {Finished} --------->

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

• This is effectively SIGMA-I

• So what if we formalize it
TLS Authentication Block

- Consists of: Certificate, CertificateVerify, Finished
  - Use this every time we want to authenticate
  - Sometimes Cert/CertVerify are omitted

- Inputs are:
  - A Session Context (usually the handshake transcript)
  - A base key to compute the finished keys from
    * Client and server use separate keys

- CertificateVerify = Sign(SC + Certificate)

- Finished = MAC(SC + Certificate + CertificateVerify)
  - Note: this is like continuing the hashes
## Authentication Inputs

<table>
<thead>
<tr>
<th>Mode</th>
<th>Handshake Context</th>
<th>Base Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-RTT</td>
<td>ClientHello + ServerConfiguration + Server Certificate + CertificateRequest</td>
<td>xSS</td>
</tr>
<tr>
<td>1-RTT (Server)</td>
<td>ClientHello ... ServerConfiguration</td>
<td>master_secret</td>
</tr>
<tr>
<td>1-RTT (Client)</td>
<td>ClientHello ... ServerFinished</td>
<td>master_secret</td>
</tr>
<tr>
<td>Post-Handshake</td>
<td>ClientHello ... ClientFinished + CertificateRequest</td>
<td>master_secret</td>
</tr>
</tbody>
</table>
Post-Handshake Client Auth

- Server can send CertificateRequest at any time
- Client responds with authentication block
  - Possibly with empty cert
- Note: need to add correlator between CertificateRequest and CertificateVerify
  - Needs to include freshness from server
  - Not in this PR yet
Key Schedule Changes

3. \( mSS = \text{HKDF-Expand-Label}(xSS, \text{"expanded static secret"}, \) 
   \( \text{handshake_hash}, L) \)

4. \( mES = \text{HKDF-Expand-Label}(xES, \text{"expanded ephemeral secret"}, \) 
   \( \text{handshake_hash}, L) \)

Where \( \text{handshake_hash} \) includes all messages up through the 
server CertificateVerify message.

5. \( \text{master_secret} = \text{HKDF-Extract}(mSS, mES) \)

\( \text{client}_{\text{finished}}_{\text{key}} = \) 
\( \text{HKDF-Expand-Label}(\text{BaseKey, "client}_{\text{finished}}", \text{""}, L) \)

\( \text{server}_{\text{finished}}_{\text{key}} = \) 
\( \text{HKDF-Expand-Label}(\text{BaseKey, "server}_{\text{finished}}", \text{""}, L) \)
ClientHello
   + ClientKeyShare
      + EarlyDataIndication
          + EarlyDataIndication

0-RTT | (Certificate*)
mode  | (CertificateVerify*)

v (Finished) // Note: new message.
      (Application Data*)

----------->

ServerHello
   ServerKeyShare*
   {EncryptedExtensions}
   {CertificateRequest*}
   {ServerConfiguration*}
      {Certificate*}
          {CertificateVerify*} | Server Auth.

-----------

v

1-RTT ^ {Certificate*}

Client | {CertificateVerify*}

Auth   | {Finished}

v [Application Data]

---------->

[Application Data]

----------

[CertificateRequest]

[Certificate] | Post-HS
[Finished]      v
Other Notes

- Added Finished to 0-RTT data
  - It’s part of authentication block
  - Adds consistency and a natural separator
- 0-RTT data isn’t hashed into transcript for 1-RTT
  - Conceptually cleaner to separate these
  - Not necessary for negotiation
- Possible to client authenticate both in 0-RTT and 1-RTT
  - Conceptually simpler
  - Server can keep requesting anyway
- We discussed merging Certificate and CertificateVerify
  - I haven’t forgotten. Stay tuned.
Framing for 0-RTT(#311, #295)

- 0-RTT content types are funny
  - Handshake uses "early_data"
  - Application uses "application_data"
- Idea was to separate by content type
  - Even without keys
- This doesn’t work with encrypted content types
- Proposed resolution
  - 0-RTT content uses the expected content types
  - Terminate 0-RTT application data with close_notify
  - Recovering from a failed 0-RTT requires trial decryption
HelloRetryRequest and Handshake Hash (#104, #185)

- Document is agnostic about handshake hash when HRR is used
- Option 1: Continue hash
  - Much easier to analyze for handshake correctness
  - But we want the HRR to be stateless
    * Combine HRR with DTLS cookie exchange
- Option 2: Reset hash
  - Easy to make stateless
  - Much harder to analyze
- It turns out we can have both good properties
Stateless HelloRetryRequest

- Import cookie exchange from DTLS
  - Server sends a cookie with HRR
  - Client echoes back cookie with new Hello

- Retain existing rules for repeat ClientHello construction
  - Append new ClientKeyShare (if needed)
  - Add cookie
  - No other changes

- Server can recover the handshake hash state
  - Option 1: offload state into cookie (integrity protected)
  - Option 2: reconstruct the ClientHello from the rules above
  - Option 3: Or just keep state (makes sense in TLS)

- This is all invisible to the client
Other cookie construction issues

- Cookie should indicate why HRR was sent
  - Needed for Option #2.
  - Can still be opaque

- Want to allow use of cookie as “address token”
  - Client can send it repeatedly
  - Do we need structure in the cookie to indicate that?
Re-Keying

• AES-GCM and ChaCha20/Poly1305 can’t encrypt infinite amounts of data

• Some debate about exactly where the boundaries are

• But potentially within plausible bounds for TLS
  – Watson Ladd recommends $2^{32}$ blocks for AES-GCM and $2^{96}$ blocks for ChaCha/Poly1305
  – David McGrew (offlist) recommends $2^{32}$ records for AES-GCM
  – For reference [draft-ietf-avtcore-srtp-aes-gcm] specifies $2^{48}$ records

• Security bounds are different for TLS and DTLS because attacker can query DTLS oracle more than once
  – DTLS could have a hard limit on failures?
Seattle Discussion Consensus on Technical Approach

- Don’t set a hard limit
  - This accommodates new results
- Have a one-way indicator that says “I am changing my key”
  - Message type should be handshake (or alert?)
  - Other side MAY (but not MUST) do the same thing
  - With DTLS also update epoch in case message is lost
Proposed Way Forward

- Determine what we consider acceptable limits
  - $X$ number of records with a $Y$ margin of safety
- Ask CFRG a targeted question about those limits with current algorithms
  - If we’re at all close, add a rekeying mechanism as above (PR wanted)
- Discuss: what are $X$ and $Y$?
Exporters for TLS 1.3 (#282)

Obvious construct:

Exporter(Label, Context, L) =
    HKDF-Expand-Label(exporter_secret, Label, Context, L)

• Important note: this doesn’t include client cert
  – But does include the server cert
  – So less context than TLS 1.2 with session hash
  – Analysis needed
TLS-Unique

• Do we still need this?
  – Applications (e.g., Tokbind) are moving to exporters
Other Issues?