cTLS

draft-rescorla-tls-ctls-03
Eric Rescorla, Richard Barnes, Hannes Tschofenig
What problems we are trying to solve?

- Legacy cruft in TLS 1.3 handshake
- Ability to have reduced profiles of TLS
  - Small wire and size for constrained applications
  - “Simple” TLS for applications which don’t need the entire feature set (e.g., 0-RTT)
- Clearer separation between handshake and record layer
  - Allow handshake to be used with other record layers (e.g., QUIC)

Many of these were issues we punted out of 1.3
Motivating Use Cases

- QUIC
- ATLS
- LAKE
- EAP
Two (and a half) technical pieces

- Clean up the handshake messages a bit
- A specialization mechanism for describing subsets of TLS
- More clearly delineate how to plug handshake into new record layers
Clean up handshake messages

- Replace all integers with varints
- Remove some unnecessary “legacy” fields
  - E.g., session_id
- Remove handshake message length from Handshake framing
  - All messages are already self-describing

One difficulty: backward compatibility
Specialization Mechanism

- TLS is a general protocol
  - But not everyone wants all the flexibility
- General idea: monomorphize along individual axes (e.g., version)
  - Nail down the value of that axis
  - Remove on-the-wire representation of the negotiation point for that axis
  - Transcript is reconstructed to include what would have been sent
- Specialization with forward-compatibility
  - Remove unneeded extensions ... but otherwise allow extensions
  - Compress known certificates ... but also allow unknown certificates
**One way of thinking about this**

<table>
<thead>
<tr>
<th>Handshake</th>
<th>Application</th>
<th>Alert</th>
</tr>
</thead>
<tbody>
<tr>
<td>cTLS Compression Layer</td>
<td></td>
<td>Profile</td>
</tr>
<tr>
<td>cTLS Record Layer / Application</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
JSON Syntax

- Specializations are defined in a JSON syntax
- Partly just a formalism
- But also provides a machine readable form so you could automatically monomorphize
- Should we define a canonical wire encoding/defined profiles, etc.?
Example: JSON Syntax

```
{
    "version" : 772,
    "cipherSuite" : "TLS_AES_128_GCM_SHA256"
}
```

- This means “do only TLS 1.3 with AES_128_GCM_SHA256”
- Omit “supported_versions” and “cipher_suites” fields on the wire
- Decompressed transcript has single-valued fields
Predefined Extensions

- Predefined extensions don’t appear on the wire
  - Generally just defined as fixed hex strings
  - But do appear in the transcript
- Otherwise extensions are encoded as usual
- All extensions have to appear in code point order
  - Except for PSK, obviously!
  - This is a change from TLS 1.3
  - ... but it’s compatible
Extended Example

{
    "version": 772,
    "cipherSuite": "TLS_AES_128_CCM_8_SHA256",
    "dhGroup": "X25519",
    "signatureAlgorithm": "ECDSA_P256_SHA256",
    "randomSize": 8,
    "finishedSize": 8,
    "clientHelloExtensions": {
        "server_name": "000e00000b6578616d706c652e636f6d",
    },
    "certificateRequestExtensions": {
        "signature_algorithms": "00020403"
    }
}
Known Certificates

- A map of certificates in hex and a short nickname
- Nickname just gets encoded in the CertificateEntry field
  - This means they need to be distinguishable from certs
  - Make them short, don’t start with 0x30, etc.
- Expanded in the transcript like everything else
# Initial Performance Numbers (short Finished, Random)

<table>
<thead>
<tr>
<th></th>
<th>ECDHE</th>
<th></th>
<th>PSK</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TLS</td>
<td>CTLS</td>
<td>Overhead</td>
<td>TLS</td>
</tr>
<tr>
<td>ClientHello</td>
<td>132</td>
<td>50</td>
<td>10</td>
<td>147</td>
</tr>
<tr>
<td>ServerHello</td>
<td>90</td>
<td>48</td>
<td>8</td>
<td>56</td>
</tr>
<tr>
<td>ServerFlight</td>
<td>478</td>
<td>104</td>
<td>16</td>
<td>42</td>
</tr>
<tr>
<td>ClientFlight</td>
<td>458</td>
<td>100</td>
<td>11</td>
<td>36</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1158</td>
<td>302</td>
<td>45</td>
<td>280</td>
</tr>
</tbody>
</table>
**STATUS**

- Adopted as work item for TLS
  - Awaiting final charter change
- Implementations in Go and Rust (in progress)
- Working to extend TLS 1.3 proofs
**Handshake/Record Layer Separation**

- These are nominally separate but actually tied together
- **QUIC** separates them
  - TLS 1.3 handshake
  - Its own record layer
- **Plan**: firm up the interface and requirements on the “record layer”
  - Really retconning what happened in QUIC