TLS 1.3: What developers should know about the APIs

Daiki Ueno
Red Hat Crypto team
TLS 1.3: RFC 8446

Published in August 2018

- Low latency
- More security
- Cleaner protocol
Low latency:
1-RTT handshake

TLS 1.2: 2-RTT

TLS 1.3: 1-RTT

application traffic

key exchange &
authentication

authentication* + client Finished
More security

• No RSA / static DH key exchange
• Legacy algorithms were removed
• All symmetric ciphers are AEAD
Protocol refactoring

- Ciphersuites
- Session resumption
Ciphersuites

Key Exchange  Cipher Algorithm  MAC Algorithm  Hash Algorithm

key_share extension

TLS 1.2

> 100

TLS 1.3

= 5
Session resumption

TLS 1.2

1\textsuperscript{st} connection
Client and server share the secret

2\textsuperscript{nd} connection
Client sends the previous session ID

Client keeps the secret

Server stores the state in the session cache
Session resumption

TLS 1.3

1\textsuperscript{st} connection
Client and server share the secret
Session ticket

Client keeps the secret

2\textsuperscript{nd} connection
Client sends back the ticket

Server doesn’t need to keep the secret

Forward secure if DHE key exchange is used
How can I use TLS 1.3?

- Enabled in major libraries
  - OpenSSL, GnuTLS, NSS

- No little code changes are needed
  - for typical use-cases

- New features need new API
New features

- Post-handshake authentication
- Key update
- Length hiding
- 0-RTT mode
What is a good API?

• Usability
• Flexibility
Usability

• Easy to use, hard to misuse

• “Huffman coding” by usage pattern
  – Less code for common use-cases
  – More code for uncommon use-cases

• Default to be safe
Flexibility

• Scale from embedded to servers
  – Decouple resource access from the code
  – Provide callbacks

• Future proof
  – There will probably be TLS 1.4
  – Don’t assume parameters are fixed
What is a good API?

Usability

Flexibility

Christoph Strässler (CC-BY-SA 2.0)
Existing design choices

• I/O abstraction
  – Generic I/O used for both TLS and non-TLS

• Handshake
  – Explicitly triggered or implicitly during I/O

• Resumption data
  – Manually or automatically tracked
## Existing design choices

<table>
<thead>
<tr>
<th></th>
<th>I/O abstraction</th>
<th>Handshake</th>
<th>Resumption data</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSSL</td>
<td>Yes (BIO)</td>
<td>Implicit / explicit</td>
<td>Automatic (cached per-ctx)</td>
</tr>
<tr>
<td>GnuTLS</td>
<td>No</td>
<td>Explicit</td>
<td>Manual</td>
</tr>
<tr>
<td>NSS</td>
<td>Yes (NSPR)</td>
<td>Implicit / explicit</td>
<td>Automatic (cached per-process)</td>
</tr>
</tbody>
</table>
New features, new API

- Post-handshake authentication
- Key update
- Length hiding
- 0-RTT mode
Post-handshake auth

- The server can request client-auth *at any time* with a CertificateRequest message
- Re-associate client’s identity with a different certificate
- Can delay client authentication until a resource is actually requested
Post-handshake auth

/* client: indicate post handshake auth */
SSL_set_post_handshake_auth(client, 1);

/* server: request post handshake auth */
SSL_verify_client_post_handshake(server);

/* client: indicate post handshake auth */
gnutls_init(&client, ...
|GNUTLS_POST_HANDSHAKE_AUTH);

/* server: request post handshake auth */
gnutls_reauth(server, 0);

Not implemented
Key update

- Peers can update traffic keys with a KeyUpdate message
- There is a limit of data that can be safely encrypted with a single key
  - GnuTLS and NSS implement automatic key updates
Key update

/* schedule key update, and request the peer to
 * update their key */
SSL_key_update(s, SSL_KEY_UPDATE_REQUESTED);

/* schedule key update, and request the peer to
 * update their key */
gnutls_session_key_update(s, GNUTLS_KU_PEER);

/* schedule key update, and request the peer to
 * update their key */
SSL_KeyUpdate(s, PR_TRUE);
Length hiding

Prevent attackers being able to guess the actual content length
/* default to pad multiple of 4096 */
SSL_set_block_padding(s, 4096);

/* override the padding with a callback per message */
SSL_set_record_padding_callback(s, padding_cb);

/* send application data */
SSL_write(s, data, size);

static size_t
padding_cb(SSL *s, int type, size_t len, void *arg)
{
    /* return new padding */
}
Length hiding

/* send application data without padding */
gnutls_record_send(s, data, size, θ);

/* send application data with arbitrary padding */
gnutls_record_send2(s, data, size, pad, θ);

Not implemented
**0-RTT mode**

**TLS 1.3: 1-RTT**
- Key exchange & authentication
- Authentication* + client Finished

**TLS 1.3: 0-RTT**
- Resuming handshake
- Early application data
- Even lower latency
0-RTT: Sending

1. Check the maximum amount of data the server would accept

2. Send early data

3. Check if the server has accepted it; otherwise re-send the data as 1-RTT
0-RTT: Sending

1. /* check the maximum data the server would accept */
   
   maxsize = 
   SSL_SESSION_get_max_early_data(SSL_get0_session(client));
   if (size > maxsize)
       return -1;

2. /* send early data before handshake */
   SSL_write_early_data(client, data, size, &written);
   /* do handshake, either explicitly or implicitly */

3. /* check if the early data was accepted */
   status = SSL_get_early_data_status(client);
   if (status != SSL_EARLY_DATA_ACCEPTED) {
       /* early data was rejected; resend it as 1-RTT */
       SSL_write(client, data, size);
   }
0-RTT: Sending

1. /* check the maximum data the server would accept */
   maxsize =
   gnutls_record_get_max_early_data_size(client);
   if (size > maxsize)
       return -1;

2. /* send early data before handshake */
   gnutls_record_send_early_data(client, data, size);
   gnutls_handshake(client);

3. /* check if the early data was accepted */
   flags = gnutls_session_get_get_flags(client);
   if (!(flags & GNUTLS_SFLAGS_EARLY_DATA)) {
       /* early data was rejected; resend it as 1-RTT */
       gnutls_record_send(client, data, size);
   }
0-RTT: Sending

1 /* enable 0-RTT */
SSL_OptionSet(client, SSL_ENABLE_0RTT_DATA, PR_TRUE);
/* check the maximum data the server would accept */
SSL_GetResumptionTokenInfo(tokenData, tokenLen,
&token, &len);
if (size > maxsize)
    return -1;

2 /* send early data before handshake */
PR_Send(client, data, size);
/* do handshake, either explicitly or implicitly */

3 /* check if the early data was accepted */
SSL_GetChannelInfo(client, &info, sizeof(info));
if (!info.earlyDataAccepted) {
    /* early data was rejected; resend it as 1-RTT */
    PR_Send(client, data, size);
}
0-RTT mode: Risks

- No forward secrecy
- Replay attacks
0-RTT: Anti-replay

- Single use tickets
  - OpenSSL

- Client Hello recording
  - GnuTLS, NSS
Single-use tickets

• Record issued session tickets in DB
• Remove ticket once it is used
  – Not limited to 0-RTT
  – Session tickets are long lived: ~1 week
Client Hello recording

- Initial session
- Ticket (T1)
- Resumption with ticket (T1)
- Resumption with ticket (T1) with same ticket age

Record ClientHello within a certain time window ~10 sec

- Client sends duplicated ClientHello

Client’s view of ticket age – Server’s view of ticket age > ~10 sec
0-RTT: Receiving

1. Enable 0-RTT with anti-replay

2. Accept or reject early data
0-RTT: Receiving

1. /* enable 0-RTT */
   SSL_set_max_early_data(server, 65535); /* optional */

   /* anti-replay mechanism is on by default,
   * implemented using server session cache */

2. /* receive early data, before application data */
   while (ret != SSL_READ_EARLY_DATA_FINISH) {
      ret = SSL_read_early_data(server, buf, sizeof(buf),
                               &readbytes);
      if (ret == SSL_READ_EARLY_DATA_SUCCESS)
         /* early data received */;
   }

   /* early data received */;
/* enable early data receiving */
gnutls_init(&server, ...|GNUTLS_ENABLE_EARLY_DATA);
/* optional */
gnutls_record_set_max_early_data_size(server, 65535);
/* set up anti-replay mechanism */
gnutls_anti_replay_init(&ar);
gnutls_anti_replay_add_function(ar, ar_add_func);
gnutls_anti_replay_set_window(ar, 10000); /* optional */
gnutls_anti_replay_enable(server, ar);

static void
ar_add_func(void *ptr, time_t exp_time,
    const gnutls_datum_t *key,
    const gnutls_datum_t *data)
{
    /* add key/data if it doesn’t exist */
}
/ * retrieve early data through a handshake hook */
gnutls_handshake_set_hook_function(server,
    GNUTLS_HANDSHAKE_END_OF_EARLY_DATA,
    handshake_hook_func);

static int
handshake_hook_func(gnutls_session_t session,
    unsigned int htype,  
    unsigned when,  
    unsigned int incoming,  
    const gnutls_datum_t *msg)
{
    ...
    return gnutls_record_recv_early_data(session,
        buf,
        sizeof(buf));
}
0-RTT: Receiving

1. /* enable early data receiving */
   SSL_OptionSet(server, SSL_ENABLE_0RTT_DATA, PR_TRUE);
   SSL_SetMaxEarlyDataSize(server, 65535); /* optional */

   /* setup anti-replay mechanism
   * NSS internally uses Bloom filters to detect dupes
   * where k = 7, bits = 14
   */
   SSL_SetupAntiReplay(10 * PR_USEC_PER_SEC, 7, 14);

2. /* receive early data as part of normal data */
   PR_Read(server, buf, sizeof(buf));
Summary

• TLS 1.3 > TLS 1.2
• TLS 1.3 also brings additional features
• Those features need new API
• API designs have *reasons* behind them
Resources

• TLS 1.3 – OpenSSLWiki

• GnuTLS and TLS 1.3

• NSS