New Handshake Flows for TLS 1.3
draft-rescorla-tls13-new-flows-00

Abstract

This document sketches some potential new handshake flows for TLS 1.3.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on May 9, 2014.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November
Table of Contents

1. Introduction ........................................... 3
2. Background ........................................... 3
   2.1. Handshake Flows ................................... 3
   2.2. Handshake Latency .................................. 4
   2.3. Plaintext Data in the Handshake .................... 5
3. Basic Assumptions ...................................... 6
4. Challenging Issues ...................................... 7
5. New Flows ............................................ 8
   5.1. Reduced RTT Handshakes w/o Server-Side State .......... 8
      5.1.1. Improving Privacy ............................... 12
         5.1.1.1. A Partially Private Handshake ................. 12
      5.1.1.2. Partially Private Fallback .................... 13
   5.2. Reduced RTT Handshakes with Server-Side State ........ 15
      5.2.1. Zero Round-Trip Resumed Handshake ............... 15
      5.2.2. Zero Round-Trip Non-Resumed Handshake .......... 15
      5.2.3. Fallback from 0-RTT handshakes .................. 16
      5.2.4. Zero Round Trip (sort-of) with PFS ............... 17
      5.2.5. Anti-Replay Options ............................. 17
   5.3. Compatibility ..................................... 18
   5.4. Some other questions to think about ................. 18
6. Backward Compatibility .................................. 18
7. Security Considerations ................................ 18
8. Acknowledgements ...................................... 18
9. References ........................................... 19
   9.1. Normative References ............................... 19
   9.2. Informative References ............................. 19
Appendix A. Summary of Existing Extensions .................. 19
Author’s Address ......................................... 21
1. Introduction

DISCLAIMER: THIS IS A VERY ROUGH DRAFT. EVERYTHING HERE IS SUPER-HANDWAVY AND HASN’T REALLY HAD ANY SECURITY ANALYSIS. I DON’T PROMISE IT’S NOT VERY VERY WRONG BUT I WANTED TO BE ABLE TO HAVE AN EARLY DISCUSSION ABOUT DIRECTION.

The TLS WG is specifying TLS 1.3, a revision to the TLS protocol. The two major design goals for TLS 1.3 are:

- Reduce the number of round trips in the handshake, providing at least "zero round-trip" mode where the client can send its first data message immediately without waiting for any response from the server.
- Encrypt as much of the handshake as possible in order to protect against monitoring of the handshake contents.

This document proposes revisions to the handshake to achieve these objectives. They are being described in a separate document and in somewhat schematic form for ease of analysis and discussion. If they are considered acceptable, some of them may be elaborated and integrated into the main TLS document.

2. Background

In this section we briefly review the properties of TLS 1.2 [RFC5246]

2.1. Handshake Flows

As a reminder, this section reproduces the two major TLS handshake variants, from [RFC5246]. For clarity, data which is cryptographically protected by the record protocol (i.e., encrypted and integrity protected) are shown in braces, as in {Finished}.
2.2. Handshake Latency

The TLS "Full Handshake" shown above incurs 2RTT of latency: the client waits for the server Finished prior to sending his first application data record. The purpose of the Finished is to allow the client to verify that the handshake has not been tampered with, for instance that the server has not mounted a downgrade attack on the cipher suite negotiation. However, if the client is satisfied with the handshake results (e.g., the server has selected the strongest parameters offered by the client), then the client can safely send its first application data immediately after its own Finished (this is often called either "False Start" or "Cut Through" [I-D.bmoeller-tls-falsestart]), thus reducing the handshake latency to 1RTT for a full handshake.
This technique is not explicitly authorized by the TLS specification but neither is it explicitly forbidden. A number of client implementations (e.g., Chrome, Firefox, and IE) already do False Start. However, because some servers fail if they receive the Finished early, it is common to use some kind of heuristic to determine whether a server is likely to fail and therefore whether this optimization can be used.

The abbreviated handshake already succeeds in 1RTT from the client’s perspective.

There have been proposals to take advantage of cached state between the client and server to reduce the handshake latency to 0RTT [I-D.agl-tls-snapstart]. However, they have not been widely adopted.

2.3. Plaintext Data in the Handshake

As shown in the figures above, essentially the entire handshake is in the clear. Some of these values are potentially sensitive, including:

- The client certificate.
- The server name indication (i.e., which server the client is trying to contact.
- The server certificate (this is only interesting when SNI is used)
o The next protocol in use [I-D.ietf-tls-applayerprotoneg].

o The channel ID [REF: Channel ID]

o The client cipher suite list (potentially usable for client fingerprinting.)

There have been proposals to address this just for extensions [I-D.agl-tls-nextprotoneg] as well as for the handshake as a whole [I-D.ray-tls-encrypted-handshake]. In general, the amount of privacy protection which can be provided is somewhat limited by four factors:

o A fair amount of information can be gathered from traffic analysis based on message size and the like.

o Because the existing mechanisms do not encrypt these values, an active attacker can generally simulate being a server which does not accept whatever new handshake protection mechanisms are offered and force the client back to the old, unprotected mechanism. This form of active attack can be mitigated by refusing to use the old mechanism, however that is not always possible if one wishes to retain backward compatibility.

o Many inspection devices mount a man-in-the-middle attack on the connection and therefore will be able to inspect information even if it is encrypted.

o It’s very hard to avoid attackers learning the server’s capabilities because they generally fall into an easily probeable/enumerable set and in most cases the clients are anonymous (and thus indistinguishable from attackers). Probably the best we can do is prevent attackers from learning which of a server’s capabilities a given client is exercising.

However, there are still advantages to providing protection against passive inspection. The flows in this document attempt to provide this service to the extent possible.

3. Basic Assumptions

This section lays out the basic assumptions that motivate the designs in this document (aside from the objectives in Section 1."

Retain Basic TLS Structure: The intent of this document is to retain the basic TLS structure and messages, tweaking them as minimally as necessary to accomplish the objectives in Section 1. Conservative design is good when working with security protocols.
Backward Compatibility is Required: It must be possible for TLS 1.3 implementations to interoperate with TLS 1.2 and below.

Minimize Variation: TLS already has a very large number of variant handshakes which makes it confusing to analyze. We would like to avoid multiplying this unnecessarily. We will probably deprecate some of the old flows in TLS 1.3.

PFS is important but not mandatory: Perfect Forward Secrecy is an important security property, but it it exists in tension with low latency (as existing mechanisms depend on contributions from both sides). Accordingly, we should provide PFS-compatible modes but not mandate them.

0-RTT modes require server-side state: The existing TLS anti-replay mechanism involves the server and client jointly contributing nonces and therefore can be stateless on the server (as long as a fresh nonce can be generated.) Any 0-RTT mode in which the client sends data along with his initial handshake message must use some other mechanism to prevent replay, and this involves the server keeping some state.

Latency is often more important than bandwidth: Because networks are getting faster but the speed of light is not, it is often more important to minimize the number of round trips than the number of bits on the wire.

Client-side computation is often cheap: In many (but not all) cases, clients can afford to do a lot of cryptographic operations very cheaply.

Clients can and should be optimistic: When we put together the previous points, we come to the conclusion that it’s OK for clients to be optimistic that things will succeed. So, for instance, it’s OK to send messages to the server that might need to be retransmitted or recomputed if the server’s state is not as expected. This is a key element of a number of round-trip reducing strategies.

4. Challenging Issues

This section previews some known challenging issues to keep in mind in the text below.
SNI privacy versus round-trips and variant server configuration: SNI is intended to alter the behavior of the server, but it also leaks information about the intended server identity. These demands are in tension. In situations where the server uses incompatible cryptography for the different SNIs (e.g., the servers use only RSA and use different keys) it is not possible to hide the SNI. In other cases, e.g., where DHE is used but authenticated with different keys, it is possible to have distinct configurations but at the cost of multiple RTs in order to first exchange keys and then to send the encrypted SNI and then respond the server’s response.

Round-trips required for PFS: It’s clearly not possible to do a 0-RTT handshake while also providing PFS. The two basic alternatives are (1) abandon PFS for 0-RTT handshake (maybe using re-handshakes on the same connection to get PFS) (2) Have a two-stage crypto exchange where the client initially uses a key generated using a cached server DH share and then and then later uses a key generated from a fresh server DH share.

5. New Flows

This document includes a number of strategies for improving latency and privacy, including:

- Move the CCS up in the handshake with respect to other messages so that more of the handshake can be encrypted, thus improving protection against passive handshake inspection.
- Allow the client to send at least some of his second flight of messages (ClientKeyExchange, Finished, CCS, etc.) together with the first flight in the full handshake, thus improving latency.
- Allow the client to send data prior to receiving the server’s messages, thus improving latency.

In addition, where prior versions of TLS generally assume that the client is totally ignorant of the server’s capabilities (e.g., certificate and supported cipher suites) we assume that the client either knows them or can make a good guess (see [REF: Fast Track]) for an early example of this.)

5.1. Reduced RTT Handshakes w/o Server-Side State

NOTE: THIS SECTION IS PEDAGOGICAL. The flows in this section are not flows we probably want to adopt, but they provide a somewhat easier introduction into what we are trying to do.

The basic idea behind doing a 1RTT handshake is that the client has
some information about the server’s likely parameters. Specifically, it believes it knows:

- The server’s preferred cipher suite.
- The server’s certificate and persistent ServerKeyExchange parameters.

For starters, let’s look at what we can accomplish if we don’t care about improving privacy.

The handshake below, modelled on Fast Track and some suggestions from Brian Smith, exploits this fact to reduce handshake latency.

```
Client                                              Server
ClientHello + CI -> ServerHello + CI
ClientKeyExchange ----------> Certificate*
                        ServerKeyExchange*
                        ServerHelloDone
                        [ChangeCipherSpec]
                        <--------           {Finished}
                        [ChangeCipherSpec]
                        {Finished}
                        [Application Data] ---------->
                        {Application Data}           <------->   {Application Data}
```

Figure 3: Full handshake (fast track-ish) [PEDAGOGICAL]

In this diagram, "CI" refers to the cached info extension [I-D.ietf-tls-cached-info] which allows the client and the server to jointly agree that the client knows the server’s parameters. Note that currently cached info includes the server’s certificate but not the server’s DH or ECDH parameters (Fast-Track included both) and would need to be modified to support this usage. However, the modification is straightforward. [[IMPORTANT PRESENTATION NOTE: We are showing the ClientKeyExchange as a separate message, but obviously this is bad news for a pre-TLS 1.3 server (see Section 5.3 on compatibility, We really need to stuff any extra messages sent with the ClientHello in an extension as done in [I-D.agl-tls-snapstart], but that makes the diagrams too clumsy. Please imagine where appropriate below.]]

This handshake allows the client and the server to establish keys in 1RTT (from the client’s perspective) and the server to start
transmitting upon receiving the client’s first message. Note that the client need not have encountered the server before, especially with ECDHE. If many servers support a common parameter set the client can just send a key valid for that parameter set, at the cost of some additional, potentially unnecessary, computation.

Obviously, there is some risk that the client is incorrect about the server’s keying material, either because it has made a mistake or the server has changed it. In this case, the server responds with a ServerHello w/o the CI extension and the client is forced back to the usual TLS 1.2-style handshake, as below.

```
Client                                               Server
ClientHello + CI                                        ServerHello
                 ClientKeyExchange            -------->          Certificate*
                      ServerHello
                      Certificate*
                      ServerKeyExchange*
                      CertificateRequest*
<--------           ServerHelloDone
Certificate*
ClientKeyExchange
CertificateVerify*
[ChangeCipherSpec]
(Finished)                             -------->          [ChangeCipherSpec]
                                          <--------           (Finished)
(Application Data)         <-------->   (Application Data)
```

Figure 4: Fallback to full handshake [PEDAGOGICAL]

This design has two major drawbacks:

- It’s not possible to do client authentication strictly in 1RT (which is why the client authentication modes are not shown in Figure 3).
- We still haven’t done anything to encrypt much of the handshake.

The ultimate reason for both of these problems is replay protection for the server. Specifically, we need the ServerHello containing the server’s Random value before we can generate the keying material needed to encrypt the handshake messages. Similarly, because the CertificateVerify incorporates a random challenge provided by the server, it can’t be generated before the CertificateRequest has been provided, which adds a second round-trip.

Let’s look at the client authentication problem first. One
possibility would be to just leave the handshake as shown above and just add the CertificateRequest in the server’s first flight and the Certificate/CertificateVerify in the client’s second flight. Unfortunately, this changes the semantics of Finished so that the server’s Finished message no longer covers the entire handshake. It’s not clear what security issues this introduces, but it seems undesirable.

A better approach seems to be to introduce another server message that just confirms the first phase of the handshake. This can have the same calculation as the Finished message, but would appear (unencrypted) in the server’s first flight, as shown below:

Client

ClientHello + CI
ClientKeyExchange

Server

ServerHello + CI
Certificate*
ServerKeyExchange*
CertificateRequest*
ServerHelloDone

Certificate*
CertificateVerify*

[ChangeCipherSpec]
(Finished)
{Application Data}

<-------- AlmostFinished

Note: this handshake is not compatible with key exchange mechanisms which require a static client-side key. However, these are not widely used in any case. It is compatible with PFS cipher suites as long as the server uses stable parameters, because the client sends his share prior to seeing the ServerKeyExchange.

[[OPEN ISSUE: Should we move the server’s CCS up to before the AlmostFinished? Doesn’t seem to add much value.]]

[[OPEN ISSUE: Need a better name for AlmostFinished.]]
5.1.1. Improving Privacy

We still have not done anything to improve the privacy of the handshake, however, it is possible to improve it a fair bit by splitting the ServerHello and moving the ServerKeyExchange and CCS messages (and hence the encryption boundary) upward in a fashion similar to that suggested by Ray [I-D.ray-tls-encrypted-handshake].

5.1.1.1. A Partially Private Handshake

Client                                               Server

| ClientHello + CI                                      | ServerHello[1] + CI |
| ClientKeyExchange                                   | ServerKeyExchange* |
|                                                    | [ChangeCipherSpec]  |
|                                                    | {ServerHello[2]}    |
|                                                    | {Certificate*}      |
|                                                    | {CertificateRequest*}|
|                                                    | {ServerHelloDone}   |

<--------     {AlmostFinished}

[ChangeCipherSpec]
(Certificate*)
(CertificateVerify*)
(Finished)
(Application Data)  <-------- (Finished)
(Application Data)  <-------- (Application Data)

Figure 6: 1RTT handshake with partial privacy

Note that this handshake contains two ServerHellos, which I have labelled ServerHello[1] and ServerHello[2]. The first ServerHello contains just the minimal information to indicate that the client’s proposed parameters are acceptable (mostly what Fast Track calls the "determining parameters") with most of the information (i.e., the server’s cert, most extensions, etc.) being in the second ServerHello. Thus, nearly all the information which we consider sensitive is encrypted, with the exception of information which appears in the client hello, which obviously cannot be protected (see below). As an example, if ALPN is in use, the clients offered list of next protocols is visible but the server’s selection is not.

[[OPEN ISSUE: Should these ServerHellos have different names/code points as in [I-D.ray-tls-encrypted-handshake] There is some redundant data in the second ServerHello (e.g., the random values),]
so that’s annoying, but it might be nice to have the same messages so that we can have it appear in a bunch of different places without inventing new message names. This is a second-order issue.]

Notably, both certificates are encrypted, but the SNI is in the clear, so a passive attacker can infer the server’s identity. This is not ideal (see below) but is a compromise to remove round trips. In addition, because the ServerKeyExchange includes a signature over the keys, an attacker can verify whether a given known certificate was used.

5.1.1.2. Partially Private Fallback

If the client is incorrect about the server’s parameters (e.g., he guesses the wrong DH parameters) then the client gets forced into a slow-path handshake, as shown in Figure 7.

```
Client                                               Server
ClientHello[1] + CI                                  -------->
ClientKeyExchange                                    ServerHello[1]
                                                        <-------- ServerKeyExchange*
ClientHello[2] + CI                                  -------->
ClientKeyExchange                                    [ChangeCipherSpec]
[ChangeCipherSpec]                                    -------->
(ClientHello[3])                                    [ChangeCipherSpec]
                                                        {ServerHello}
                                                        {Certificate*}
                                                        {ServerKeySignature*}
                                                        {CertificateRequest*}
                                                        {ServerHelloDone}
                                                        <-------- {AlmostFinished}
                                                        {Certificate*}
                                                        {CertificateVerify*}
                                                        {Finished}
                                                        {Application Data} -------->
                                                        {Finished}
                                                        {Application Data} -------->
                                                        {Application Data}
```

Figure 7: Falling back from 1RTT handshake with partial privacy

This should be familiar from Figure 4: the client sends a ClientHello but incorrectly predicts the server’s parameters; the server responds with a correction indicating the parameters he wants to negotiate. This also includes a ServerKeyExchange to carry the
server’s DHE/ECDHE public key (i.e., not just the parameters). Note that the SKE must be unsigned because otherwise that reveals the server’s certificate. At this point, the client adopts the new information about the server’s state and sends a ClientHello[2] which hopefully has a more accurate prediction. [[OPEN ISSUE: Should we let this cycle repeat again if the client is still somehow wrong? Probably generate an alert.]]

As before, the client can now send a ClientKeyExchange, but because the client now has the server’s public keying material, at this point it has established a joint key which it can use for encryption. It can therefore send a ClientHello[3] with the extensions information it considers sensitive (see Appendix A.) [[OPEN ISSUE: Again, we should consider giving these different names, though here ClientHello[1] and ClientHello[2] probably should both be ClientHellos and ClientHello[3] is mostly extensions.]]

The server responds with his true second flight which is mostly encrypted but otherwise mostly as normal. The major change here is that it now includes a ServerKeySignature which covers the ServerKeyExchange from the first flight.

Note that we’ve actually killed two birds with one stone here. First, we’ve dealt with the case where the client is incorrect about the server’s capabilities, providing ourself with a fallback mechanism. Second, we’ve provided privacy for nearly the entire handshake. Thus, a client which wishes to conceal the extensions in his handshake just sends a ClientHello with a non-matching CI extension, thus eliciting the server’s key, and can then use that to encrypt the rest of the handshake.

There is one important limitation to mention regarding the use of SNI here. The handshake described in this section only works if all of the SNI values served by the server can use the same public key; either because they have the same RSA public key for all of their certificates and static RSA is used (not clear how likely this is) or because they all share the same DH/ECDH parameters (this seems reasonably likely). In cases where different SNI values require different determining parameters, then it is not possible to protect SNI and the server must somehow indicate that the client should simply retry with an ordinary handshake with SNI. There are probably a number of ways to indicate this, but (for instance) we could invent a new extension value or nonfatal alert (recall that this is only needed once the client has indicated TLS 1.3 so there is no compatibility issue.)
5.2. Reduced RTT Handshakes with Server-Side State

Without server-side state, the best-case scenario is one round trip. If we are willing to assume that the server has state, we can shorten this to zero round trips (though at the cost of PFS). The general principle is that each client handshake contains a unique value (e.g., the ClientRandom) and the server memorizes a list of which handshakes have occurred and thus can detect replay by simply checking to see if a given handshake is a replay or not. This idea appears in (for instance) [I-D.agl-tls-snapstart]. See Section 5.2.5 for some options here.

If everything goes well (the server has not changed its configuration, rebooted and lost its state, etc.) the client can follow its initial message flight (ClientHello, etc.) with encrypted data which the server will process. However, as with the handshakes in Section 5.1, there is some possibility that the server will reject the client’s parameters, in which case the client will need to fall back to the slow-path.

5.2.1. Zero Round-Trip Resumed Handshake

The resumed fast-path flow is shown below.

```
Client                                               Server

ClientHello + CI + AR [ChangeCipherSpec] (Finished) (Application Data) -------->
                        ServerHello + CI + AR [ChangeCipherSpec] (Finished) (Application Data) <--------

Figure 8: Zero Round-trip resumed handshake
```

The new "AR" field indicates an anti-replay indicator. See Section 5.2.5 for details.

5.2.2. Zero Round-Trip Non-Resumed Handshake

In the handshake shown in Figure 8, because the client’s SessionID (or SessionTicket) is carried in the clear, it is possible to link multiple handshakes initiated by the same client. This obviously isn’t an ideal property and can be addressed by not doing resumption and instead doing a 0-RTT key exchange.
The idea here is that the server’s public key (i.e., not just the DH/ECDH parameters) is reasonably static (either it’s an RSA key or a reasonably stable DHE key). The client can then send a ClientKeyExchange message which, combined with the stateful anti-replay information, is enough to set up the entire connection, as shown in Figure 9.

Client                                               Server

ClientHello[1] + CI + AR
ClientKeyExchange
{ClientHello[2]}
[ChangeCipherSpec]
{Certificate*}
{CertificateVerify*}
{Finished}
{Application Data} --------> {ServerHello[1]}
[ChangeCipherSpec]
{ServerHello[2]}
{ServerHelloDone}
<-------- {Finished}
{Application Data} <-------> {Application Data}

Figure 9: Zero Round-Trip non-resumed handshake

NOTE 1: This (potentially) includes the client’s Certificate and CertificateVerify because the client still needs to prove himself to the server, but the server doesn’t need to prove himself to the client since the client has memorized the server’s information.

NOTE 2: As usual, we need the client’s Finished message to cover the server’s expected messages. We will need to use some Snap Start-style trick to have the client include that. I believe we can solve this by shoving it in the CI, but the details need to be worked out.

If the client’s predictions of the server data are incorrect, then you fall back as in previous sections.

5.2.3. Fallback from 0-RTT handshakes

In either of the handshakes shown in the two preceding sections, there is a risk that the client will be wrong about the server’s parameters. The question then becomes how aggressive we want to be about fallback. One attitude would be to say that we want minimum protocol complexity and that therefore we want to fall back to the slowest possible path, namely the 2-RTT handshake in Figure 7.
The other attitude is to say that we want things to be as fast as possible, in which case we might want to fall back from the resumed 0-RTT handshake in Figure 8 to the non-resumed 0-RTT handshake in Figure 9 to the 1-RTT handshake in Figure 6 or to Figure 9, and then only to the 2-RTT handshake in Figure 7. This should be technically possible, at the cost of the client having to optimistically include a pile of extra stuff and having some way to phrase "use this data in this condition" in the protocol. On the other hand, it’s also obviously pretty complicated.

5.2.4. Zero Round Trip (sort-of) with PFS

Obviously, the handshake shown in Figure 9 does not provide PFS. This can be partly worked around by having the server use short-lived DH keys, but at the cost of increasing the number of times you fall back to a slower handshake (because the client is wrong about the key.) Obviously if the client wants total PFS, he can just initiate one of the 1RTT handshakes, but there are two main options for getting PFS for some of the data but not all:

- Do a complete rehandshake with a full DH exchange immediately after the resumed handshake. This gives you PFS coverage for some of the data but at some not-entirely-deterministic time.
- Have the client do two key exchanges, one with the semi-stable server key and one with a new key supplied by the server in his response. So, in Figure 9 we would add a ServerKeyExchange. The server’s application data would be encrypted with the new PFS key and the client’s application data after receiving the server’s response would be as well. However, the client’s initial data would be encrypted with a non-PFS key. [[OPEN ISSUE: how does the server determine what is what? Trial decryption? Client sentinel message?]]

5.2.5. Anti-Replay Options

There are a number of potential mechanisms for detecting replay on the server side. This section lists the main ones I know of:

- Memorize arbitrary client values, typically within some scope, (as in [I-D.agl-tls-snapstart])
- Force the client to use an identifier + values in some ordered way (e.g., by using a sliding anti-replay bitmask)
- Give the client some finite number of tokens that you remember until they are used (as an enhancement, these could be DH public keys)

Each of these approaches has costs and benefits; generally the exact form of the anti-replay mechanism is orthogonal to the design of
0-RTT handshake.

5.3. Compatibility

The high-order compatibility issue for any change to TLS is whether the ClientHello is compatible with old servers. To a first order, we deal with this by putting any new information in extensions. This is ugly, but mostly works.

However, a secondary problem becomes that some servers/accelerators are unhappy with large ClientHello messages (see https://www.imperialviolet.org/2013/10/07/f5update.html). As we optimistically put more stuff in the ClientHello, including keys and even data, we are likely to start running into problems here. Generally, we can avoid putting data into the ClientHello unless the server has indicated somehow that it is able to accept it, which should help minimize problems except if people’s configurations change or they have truly broken/mixed configurations. Nevertheless, we need to take care here or else decide we are willing to deal with some level of incompatibility.

5.4. Some other questions to think about

[TODO]

- Should we get rid of resumption?
- Should we get rid of renegotiation?
- Should we get rid of static RSA entirely?

6. Backward Compatibility

[TODO]

7. Security Considerations

Everything in this document needs further analysis to determine if it is OK.

8. Acknowledgements

This document borrows ideas from a large number of people, including Dan Boneh, Wan-Teh Chang, Cullen Jennings, Adam Langley, Nagendra Modadugu, Bodo Moeller, Marsh Ray, Hovav Shacham, Martin Thomson, and Brian Smith.
9. References

9.1. Normative References

[I-D.ietf-tls-cached-info]


9.2. Informative References

[I-D.agl-tls-nextprotoneg]

[I-D.agl-tls-snapstart]

[I-D.bmoeller-tls-falsestart]

[I-D.ietf-tls-applayerprotoneg]

[I-D.ray-tls-encrypted-handshake]

Appendix A. Summary of Existing Extensions

Many of the flows in this document have Extensions appear both both
in the clear and encrypted. This section attempts to provide a first-cut proposal (in table form) of which should appear where.

<table>
<thead>
<tr>
<th>Extension</th>
<th>Privacy level</th>
<th>Proposed Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>server_name</td>
<td>Medium</td>
<td>Server chooses (*)</td>
</tr>
<tr>
<td>max_fragment_length</td>
<td>Low</td>
<td>Clear</td>
</tr>
<tr>
<td>client_certificate_url</td>
<td>High</td>
<td>Encrypted</td>
</tr>
<tr>
<td>truncated_hmac</td>
<td>Low</td>
<td>Clear</td>
</tr>
<tr>
<td>status_request</td>
<td>Low</td>
<td>Encrypted</td>
</tr>
<tr>
<td>user_mapping</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>client_authz</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>server_authz</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>cert_type</td>
<td>Low</td>
<td>Server chooses (*)</td>
</tr>
<tr>
<td>elliptic_curves</td>
<td>Low</td>
<td>Clear</td>
</tr>
<tr>
<td>ec_point_formats</td>
<td>Low</td>
<td>Clear</td>
</tr>
<tr>
<td>srp</td>
<td>High</td>
<td>???</td>
</tr>
<tr>
<td>signature_algorithms</td>
<td>Low</td>
<td>Clear</td>
</tr>
<tr>
<td>use_srtp</td>
<td>Low</td>
<td>Encrypted</td>
</tr>
<tr>
<td>heartbeat</td>
<td>Low</td>
<td>Encrypted</td>
</tr>
<tr>
<td>alpn</td>
<td>Medium</td>
<td>Encrypted</td>
</tr>
<tr>
<td>status_request_v2</td>
<td>Low</td>
<td>Encrypted</td>
</tr>
<tr>
<td>signed_certificate_timetamp</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>SessionTicket TLS</td>
<td>Medium</td>
<td>Encrypted/Clear ***</td>
</tr>
<tr>
<td>renegotiation_info</td>
<td>Low</td>
<td>Encrypted</td>
</tr>
</tbody>
</table>

* The server may need these in the clear but the client would prefer them encrypted.

*** The SessionTicket must appear in the clear when resuming but can be encrypted when being set up.

The general principle here is that things should be encrypted where possible; extensions generally are proposed to be in the Clear part of handshake only if it seems they must be there to make the rest of the handshake work. The things that seem problematic are those which leak information about the client’s dynamic state (as opposed to implementation fingerprinting) but are potentially needed by the server for ciphersuite selection. These are labelled "Server Chooses" in this table.

It’s possible we can make life somewhat simpler by deprecating some unused extensions, but based on the table above, it looks like the extensions that make life complicated are not the ones that can be easily removed.
Author’s Address

Eric Rescorla
Mozilla
2064 Edgewood Drive
Palo Alto, CA 94303
USA

Phone: +1 650 678 2350
Email: ekr@rtfm.com