TLS Working Group
Internet-Draft
Intended status: Informational
Expires: May 7, 2020

E. Rescorla
Mozilla
R. Barnes
Cisco
H. Tschofenig
Arm Limited

November 04, 2019

Compact TLS 1.3
draft-rescorla-tls-ctls-03

Abstract

This document specifies a "compact" version of TLS 1.3. It is isomorphic to TLS 1.3 but saves space by trimming obsolete material, tighter encoding, and a template-based specialization technique. cTLS is not directly interoperable with TLS 1.3, but it should eventually be possible for a cTLS/TLS 1.3 server to exist and successfully interoperate.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

DISCLAIMER: This is a work-in-progress draft of cTLS and has not yet seen significant security analysis, so could contain major errors. It should not be used as a basis for building production systems.

This document specifies a "compact" version of TLS 1.3 [RFC8446]. It is isomorphic to TLS 1.3 but designed to take up minimal bandwidth. The space reduction is achieved by four basic techniques:

- o Omitting unnecessary values that are a holdover from previous versions of TLS.
- o Omitting the fields and handshake messages required for preserving backwards-compatibility with earlier TLS versions.
- o More compact encodings, omitting unnecessary values.
- o A template-based specialization mechanism that allows for the creation of application specific versions of TLS that omit unnecessary valuses.

For the common (EC)DHE handshake with pre-established certificates, cTLS achieves an overhead of 45 bytes over the minimum required by the cryptovariables. For a PSK handshake, the overhead is 21 bytes. Annotated handsdhake transcripts for these cases can be found in Appendix A.

Because cTLS is semantically equivalent to TLS, it can be viewed either as a related protocol or as a compression mechanism. Specifically, it can be implemented by a layer between the TLS handshake state machine and the record layer.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Structure definitions listed below override TLS 1.3 definitions; any PDU not internally defined is taken from TLS 1.3 except for replacing integers with varints.

3. Common Primitives

3.1. Varints

cTLS makes use of variable-length integers in order to allow a wide integer range while still providing for a minimal encoding. The width of the integer is encoded in the first two bits of the field as follows, with xs indicating bits that form part of the integer.

Bit pattern	Length (bytes)
0xxxxxxx	1
10xxxxxx xxxxxxxx	2
11xxxxxx xxxxxxxx xxxxxxxx	3

Thus, one byte can be used to carry values up to 127.

In the TLS syntax variable integers are denoted as "varint" and a vector with a top range of a varint is denoted as:

```
opaque foo<1..V>;
```

With a few exceptions, cTLS replaces every integer in TLS with a varint.

3.2. Record Layer

The cTLS Record Layer assumes that records are externally framed (i.e., that the length is already known because it is carried in a UDP datagram or the like). Depending on how this was carried, you might need another byte or two for that framing. Thus, only the type byte need be carried and TLSPlaintext becomes:

```
struct {
    ContentType type;
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;
```

In addition, because the epoch is known in advance, the dummy content type is not needed for the ciphertext, so TLSCiphertext becomes:

```
      struct {
          opaque content[TLSPlaintext.length];
          ContentType type;
          uint8 zeros[length_of_padding];
      } TLSInnerPlaintext;

      struct {
          opaque encrypted_record[TLSCiphertext.length];
      } TLSCiphertext;
```

Note: The user is responsible for ensuring that the sequence numbers/ nonces are handled in the usual fashion.

3.3. Handshake Layer

The cTLS handshake framing is same as the TLS 1.3 handshake framing, except for two changes:

1. The length field is omitted

2. The HelloRetryRequest message is a true handshake message instead of a specialization of ServerHello.

```
      struct {
          HandshakeType msg_type;    /* handshake type */
          select (Handshake.msg_type) {
              case client_hello:          ClientHello;
              case server_hello:          ServerHello;
              case hello_retry_request:   HelloRetryRequest;
              case end_of_early_data:     EndOfEarlyData;
              case encrypted_extensions:  EncryptedExtensions;
              case certificate_request:   CertificateRequest;
              case certificate:           Certificate;
              case certificate_verify:    CertificateVerify;
              case finished:              Finished;
              case new_session_ticket:    NewSessionTicket;
              case key_update:            KeyUpdate;
          };
      } Handshake;
```

3.4. Extensions

cTLS Extensions are the same as TLS 1.3 extensions, except varint length coded:

```
      struct {
          varint extension_type;
          opaque extension_data<0..V>;
      } Extension;
```

4. Handshake Messages

In general, we retain the basic structure of each individual TLS handshake message. However, the following handshake messages have been modified for space reduction and cleaned up to remove pre TLS 1.3 baggage.

4.1. ClientHello

The cTLS ClientHello is as follows.

```
        opaque Random[RandomLength];      // variable length

        struct {
            Random random;
            CipherSuite cipher_suites<1..V>;
            Extension extensions<1..V>;
        } ClientHello;
```

4.1.1. KeyShare, SupportedGroups, and SignatureAlgorithms

KeyShare, SupportedGroups, and SignatureAlgorithms are identical to in TLS 1.3, except for the use of varints instead of integers. Note that because all of the EC DH groups are below 0x80, they will fit into a single byte. This is not true for signature algorithms.

[[OPEN ISSUE: Should we map signature algorithms into a smaller space?]]

4.1.2. PreSharedKeys

PreSharedKeys is the same as in TLS 1.3, except for the use of varints instead of integers.

[[OPEN ISSUE: Limiting this to one value would potentially save some bytes here, at the cost of generality.]]

4.2. ServerHello

We redefine ServerHello in a similar way:

```
      struct {
          Random random;
          CipherSuite cipher_suite;
          Extension extensions<1..V>;
      } ServerHello;
```

4.3. EncryptedExtensions

Likewise, EncryptedExtensions now uses a varint length field.

```
      struct {
          Extension extensions<0..V>;
      } EncryptedExtensions;
```

[[OPEN ISSUE: We could save 2 bytes in handshake header by omitting this value when it's unneeded.]]

4.4. CertificateRequest

This message uses varint lengths and re-encodes the extensions.

```
      struct {
          opaque certificate_request_context<0..V>
          Extension extensions<1..V>;
      } CertificateRequest;
```

4.5. Certificate

We can slim down the Certficate message somewhat.

```
      enum {
          X509(0),
          RawPublicKey(2),
          (255)
      } CertificateType;

      struct {
          select (certificate_type) {
              case RawPublicKey:
                /* From RFC 7250 ASN.1_subjectPublicKeyInfo */
                opaque ASN1_subjectPublicKeyInfo<1..V>;

              case X509:
                opaque cert_data<1..V>;
          };
          Extension extensions<0..V>;
      } CertificateEntry;

      struct {
          opaque certificate_request_context<0..V>
          CertificateEntry certificate_list<1..V>;
      } Certificate;
```

4.6. CertificateVerify

This just removes the length field. ~~~~ struct { SignatureScheme algorithm; // TODO - define one byte schemes? opaque signature<1..V>; } CertificateVerify; ~~~~

4.7. Finished

Unchanged.

4.8. HelloRetryRequest

The HelloRetryRequest has the following format:

```
      struct {
          CipherSuite cipher_suite;
          Extension extensions<2..V>;
      } HelloRetryRequest;
```

It is the same as the ServerHello above but without the unnecessary sentinel Random value.

5. Template-Based Specialization

The protocol in the previous section is fully general and isomorphic to TLS 1.3; effectively it's just a small cleanup of the wire encoding to match what we might have done starting from scratch. It achieves some compaction, but only a modest amount. cTLS also includes a mechanism for achieving very high compaction using template-based specialization.

The basic idea is that we start with the basic TLS 1.3 handshake, which is fully general and then remove degrees of freedom, eliding parts of the handshake which are used to express those degrees of freedom. For example, if we only support one version of TLS, then it is not necessary to have version negotiation and the supported_versions extension can be omitted.

Importantly, this process is performed only for the wire encoding but not for the handshake transcript. The result is that the transcript for a specialized cTLS handshake is the same as the transcript for a TLS 1.3 handshake with the same features used.

One way of thinking of this is as if specialization is a stateful compression layer between the handshake and the record layer:

```
+---------------+---------------+---------------+
|   Handshake   |  Application  |     Alert     |
+---------------+---------------+---------------+    +---------+
|               cTLS Compression Layer          |<---| Profile |
+---------------+---------------+---------------+    +---------+
|          cTLS Record Layer / Application      |
+---------------+---------------+---------------+
```

Specializations are defined by a "compression profile" that specifies what features are to be optimized out of the handshake. In the following subsections, we define the structure of these profiles, and how they are used in compressing and decompressing handshake messages.

[[OPEN ISSUE: Do we want to have an explicit cTLS extension indicating that cTLS is in use and which specialization is in use? This goes back to whether we want the use of cTLS to be explicit.]]

5.1. Specifying a Specialization

A compression profile defining of a specialized version of TLS is defined using a JSON dictionary. Each axis of specialization is a key in the dictionary. [[OPEN ISSUE: If we ever want to serialize this, we'll want to use a list instead.]].

For example, the following specialization describes a protocol with a single fixed version (TLS 1.3) and a single fixed cipher suite (TLS_AES_128_GCM_SHA256). On the wire, ClientHello.cipher_suites, ServerHello.cipher_suites, and the supported_versions extensions in the ClientHello and ServerHello would be omitted.

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

cTLS allows specialization along the following axes:

version (integer): indicates that both sides agree to the single TLS version specified by the given integer value (772 == 0x0304 for TLS 1.3). The supported_versions extension is omitted from ClientHello.extensions and reconstructed in the transcript as a single-valued list with the specified value. The supported_versions extension is omitted from ClientHello.extensions and reconstructed in the transcript with the specified value.

cipherSuite (string): indicates that both sides agree to the single named cipher suite, using the "TLS_AEAD_HASH" syntax defined in [RFC8446], Section 8.4. The ClientHello.cipher_suites field is omitted and reconstructed in the transcript as a single-valued list with the specified value. The server_hello.cipher_suite field is omitted and reconstructed in the transcript as the specified value.

dhGroup (string): specifies a single DH group to use for key establishment. The group is listed by the code point name in [RFC8446], Section 4.2.7. (e.g., x25519). This implies a literal "supported_groups" extension consisting solely of this group.

signatureAlgorithm (string): specifies a single signature scheme to use for authentication. The group is listed by the code point name in [RFC8446], Section 4.2.7. (e.g., ed25519). This implies a literal "signature_algorithms" extension consisting solely of this group.

randomSize (integer): indicates that the ClientHello.Random and ServerHello.Random values are truncated to the given values. When the transcript is reconstructed, the Random is padded to the right with 0s and the anti-downgrade mechanism in {{RFC8446)}, Section 4.1.3 is disabled. IMPORTANT: Using short Random values can lead to potential attacks. When Random values are shorter than 8 bytes, PSK-only modes MUST NOT be used, and each side MUST

use fresh DH ephemerals. The Random length MUST be less than or equal to 32 bytes.

clientHelloExtensions (predefined extensions): Predefined ClientHello extensions, see {predefined-extensions}

serverHelloExtensions (predefined extensions): Predefined ServerHello extensions, see {predefined-extensions}

encryptedExtensions (predefined extensions): Predefined EncryptedExtensions extensions, see {predefined-extensions}

certRequestExtensions (predefined extensions): Predefined CertificateRequest extensions, see {predefined-extensions}

knownCertificates (known certificates): A compression dictionary for the Certificate message, see {known-certs}

finishedSize (integer): indicates that the Finished value is to be truncated to the given length. When the transcript is reconstructed, the remainder of the Finished value is filled in by the receiving side. [[OPEN ISSUE: How short should we allow this to be? TLS 1.3 uses the native hash and TLS 1.2 used 12 bytes. More analysis is needed to know the minimum safe Finished size. See [RFC8446]; Section E.1 for more on this, as well as https://mailarchive.ietf.org/arch/msg/tls/TugB5ddJu3nYg7chcyeIyUqWSbA.]]

5.1.1. Requirements on the TLS Implementation

To be compatible with the specializations described in this section, a TLS stack needs to provide two key features:

If specialization of extensions is to be used, then the TLS stack MUST order each vector of Extension values in ascending order according to the ExtensionType. This allows for a deterministic reconstruction of the extension list.

If truncated Random values are to be used, then the TLS stack MUST be configurable to set the remaining bytes of the random values to zero. This ensures that the reconstructed, padded random value matches the original.

If truncated Finished values are to be used, then the TLS stack MUST be configurable so that only the provided bytes of the Finished are verified, or so that the expected remaining values can be computed.

5.1.2. Predefined Extensions

Extensions used in the ClientHello, ServerHello, EncryptedExtensions, and CertificateRequest messages can be "predefined" in a compression profile, so that they do not have to be sent on the wire. A predefined extensions object is a dictionary whose keys are extension names specified in the TLS ExtensionTypeRegistry specified in [RFC8446]. The corresponding value is a hex-encoded value for the ExtensionData field of the extension.

When compressing a handshake message, the sender compares the extensions in the message being compressed to the predefined extensions object, applying the following rules:

- o If the extensions list in the message is not sorted in ascending order by extension type, it is an error, because the decompressed message will not match.

- o If there is no entry in the predefined extensions object for the type of the extension, then the extension is included in the compressed message

- o If there is an entry:

  - * If the ExtensionData of the extension does not match the value in the dictionary, it is an error, because decompression will not produce the correct result.

  - * If the ExtensionData matches, then the extension is removed, and not included in the compressed message.

When decompressing a handshake message the receiver reconstitutes the original extensions list using the predefined extensions:

- o If there is an extension in the compressed message with a type that exists in the predefined extensions object, it is an error, because such an extension would not have been sent by a sender with a compatible compression profile.

- o For each entry in the predefined extensions dictionary, an extension is added to the decompressed message with the specified type and value.

- o The resulting vector of extensions MUST be sorted in ascending order by extension type.

Note that the "version", "dhGroup", and "signatureAlgorithm" fields in the compression profile are specific instances of this algorithm for the corresponding extensions.

[[OPEN ISSUE: Are there other extensions that would benefit from special treatment, as opposed to hex values.]]

5.1.3. Known Certificates

Certificates are a major contributor to the size of a TLS handshake. In order to avoid this overhead when the parties to a handshake have already exchanged certificates, a compression profile can specify a dictionary of "known certificates" that effectively acts as a compression dictionary on certificates.

A known certicates object is a JSON dictionary whose keys are strings containing hex-encoded compressed values. The corresponding values are hex-encoded strings representing the uncompressed values. For example:

```
{
  "00": "3082...",
  "01": "3082...",
}
```

When compressing a Certificate message, the sender examines the cert_data field of each CertificateEntry. If the cert_data matches a value in the known certificates object, then the sender replaces the cert_data with the corresponding key. Decompression works the opposite way, replacing keys with values.

Note that in this scheme, there is no signaling on the wire for whether a given cert_data value is compressed or uncompressed. Known certificates objects SHOULD be constructed in such a way as to avoid a uncompressed object being mistaken for compressed one and erroneously decompressed. For X.509, it is sufficient for the first byte of the compressed value (key) to have a value other than 0x30, since every X.509 certificate starts with this byte.

6. Examples

The following section provides some example specializations.

TLS 1.3 only:

```
{
   "Version" : 0x0304
}
```

TLS 1.3 with AES_GCM and X25519 and ALPN h2, short random values, and everything else is ordinary TLS 1.3.

```
{
   "Version" : 772,
   "Random": 16,
   "CipherSuite" : "TLS_AES_128_GCM_SHA256",
   "DHGroup": "X25519",
   "Extensions": {
      "named_groups": 29,
      "application_layer_protocol_negotiation" : "030016832",
      "..." : null
    }
}
```

Version 772 corresponds to the hex representation 0x0304, named group "29" (0x001D) represents X25519.

[[OPEN ISSUE: Should we have a registry of well-known profiles?]]

7. Security Considerations

WARNING: This document is effectively brand new and has seen no analysis. The idea here is that cTLS is isomorphic to TLS 1.3, and therefore should provide equivalent security guarantees.

The use of key ids is a new feature introduced in this document, which requires some analysis, especially as it looks like a potential source of identity misbinding. This is, however, entirely separable from the rest of the specification.

Transcript expansion also needs some analysis and we need to determine whether we need an extension to indicate that cTLS is in use and with which profile.

8. IANA Considerations

This document has no IANA actions.

9. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

Appendix A. Sample Transcripts

In this section, we provide annotated example transcripts generated using a draft implementation of this specification in the mint TLS library. The transcripts shown are with the revised message formats defined above, as well as specialization to the indicated cases, using the aggressive compression profiles noted below. The resulting byte counts are as follows:

	ECDHE			PSK		
	TLS	CTLS	Overhead	TLS	CTLS	Overhead
ClientHello	132	50	10	147	67	15
ServerHello	90	48	8	56	18	2
ServerFlight	478	104	16	42	12	3
ClientFlight	458	100	11	36	10	1
Total	1158	302	45	280	107	21

To increase legibility, we show the plaintext bytes of handshake messages that would be encrypted and shorten some of the cryptographic values (shown with "..."). The totals above include 9 bytes of encryption overhead for the client and server flights, which would otherwise be encrypted (with a one-byte content type and an 8-byte tag).

Obviously, these figures are very provisional, and as noted at several points above, there are additional opportunities to reduce overhead.

NOTE: We are using a shortened Finished message here. See Section 5.1 for notes on Finished size. However, the overhead is constant for all reasonable Finished sizes.]]

A.1. ECDHE and Mutual Certificate-based Authentication

Compression Profile:

```
   {
     "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"
     },
     "knownCertificates": {
       "61": "3082...",
       "62": "3082..."
     }
   }
```

```
   ClientHello: 50 bytes = RANDOM(8) + DH(32) + Overhead(10)

   01                     // ClientHello
   2ef16120dd84a721       // Random
   28                     // Extensions.length
   33 26                  // KeyShare
     0024                 // client_shares.length
       001d               // KeyShareEntry.group
       0020 a690...af948 // KeyShareEntry.key_exchange

   ServerHello: 48 = RANDOM(8) + DH(32) + Overhead(8)

   02                  // ServerHello
   962547bba5e00973   // Random
   26                  // Extensions.length
   33 24               // KeyShare
     001d              // KeyShareEntry.group
     0020 9fbc...0f49 // KeyShareEntry.key_exchange

   Server Flight: 96 = SIG(71) + MAC(8) + CERTID(1) + Overhead(16)
```

```
   08                  // EncryptedExtensions
     00                //   Extensions.length
   0d                  // CertificateRequest
     00                //   CertificateRequestContext.length
     00                //   Extensions.length
   0b                  // Certificate
     00                //   CertificateRequestContext
     03                //   CertificateList
       01              //     CertData.length
         61            //       CertData = 'a'
       00              //   Extensions.length
   0f                  // CertificateVerify
     0403              //   SignatureAlgorithm
     4047 3045...10ce //   Signature
   14                  // Finished
     bfc9d66715bb2b04 //   VerifyData
```

Client Flight: 91 bytes = SIG(71) + MAC(8) + CERTID(1) + Overhead(11)

```
   0b                  // Certificate
     00                //   CertificateRequestContext
     03                //   CertificateList
       01              //     CertData.length
         62            //       CertData = 'b'
       00              //     Extensions.length
   0f                  // CertificateVerify
     0403              //   SignatureAlgorithm
     4047 3045...f60e //   Signature.length
   14                  // Finished
     35e9c34eec2c5dc1 //   VerifyData
```

A.2. PSK

Compression Profile:

```
{
  "version": 772,
  "cipherSuite": "TLS_AES_128_CCM_8_SHA256",
  "signatureAlgorithm": "ECDSA_P256_SHA256",
  "randomSize": 16,
  "finishedSize": 0,
  "clientHelloExtensions": {
    "server_name": "000e00000b6578616d706c652e636f6d",
    "psk_key_exchange_modes": "0100"
  },
  "serverHelloExtensions": {
    "pre_shared_key": "0000"
  }
}
```

ClientHello: 67 bytes = RANDOM(16) + PSKID(4) + BINDER(32) + Overhead(15)

```
01                                // ClientHello
e230115e62d9a3b58f73e0f2896b2e35 // Random
2d                                // Extensions.length
29 2b                             // PreSharedKey
    000a                          //   identities.length
      0004 00010203               //     identity
      7bd05af6                    //     obfuscated_ticket_age
    0021                          //   binders.length
      20 2428...bb3f              //     binder
```

ServerHello: 18 bytes = RANDOM(16) + 2

```
02                                 // ServerHello
7232e2d3e61e476b844d9c1f6a4c868f  // Random
00                                 // Extensions.length
```

Server Flight: 3 bytes = Overhead(3)

```
08    // EncryptedExtensions
  00  //   Extensions.length
14    // Finished
```

Client Flight: 1 byte = Overhead(3)

```
14    // Finished
```

Acknowledgments

We would like to thank Karthikeyan Bhargavan, Owen Friel, Sean Turner, Martin Thomson and Chris Wood.

Authors' Addresses

Eric Rescorla
Mozilla

Email: ekr@rtfm.com

Richard Barnes
Cisco

Email: rlb@ipv.sx

Hannes Tschofenig
Arm Limited

Email: hannes.tschofenig@arm.com