Workgroup: TLS Working Group
Internet-Draft: draft-ietf-tls-ctls-04
Published: 25 October 2021
Intended Status: Standards Track
Expires: 28 April 2022
Authors: E. Rescorla   R. Barnes   H. Tschofenig
         Mozilla   Cisco   Arm Limited

Compact TLS 1.3

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, a template-based specialization technique, and alternative cryptographic techniques. 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

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 https://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 28 April 2022.

Copyright Notice

Copyright (c) 2021 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 (https://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.

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 five basic techniques:

* Omitting unnecessary values that are a holdover from previous versions of TLS.

* Omitting the fields and handshake messages required for preserving backwards-compatibility with earlier TLS versions.

* More compact encodings, for example point compression.

* A template-based specialization mechanism that allows pre-populating information at both endpoints without the need for negotiation.

* Alternative cryptographic techniques, such as semi-static Diffie-Hellman.

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 handshake 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.

2.1. Template-based Specialization

A significant transmission overhead in TLS 1.3 is contributed to by two factors, : - the negotiation of algorithm parameters, and extensions, as well as - the exchange of certificates.

TLS 1.3 supports different credential types and modes that are impacted differently by a compression scheme. For example, TLS supports certificate-based authentication, raw public key-based authentication as well as pre-shared key (PSK)-based authentication. PSK-based authentication can be used with externally configured PSKs or with PSKs established through tickets.

The basic idea of template-based specialization 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     |
+---------------+--------------+--------------+
```

By assuming that out-of-band agreements took place already prior to the start of the cTLS protocol exchange, the amount of data exchanged can be radically reduced. Because different clients may use different compression templates and because multiple compression templates may be available for use in different deployment environments, a client needs to inform the server about the profile it is planning to use. The profile field in the ClientHello serves this purpose.

Although the template-based specialization mechanisms described here are general, we also include specific mechanism for certificate-based exchanges because those are where the most complexity and size reduction can be obtained. Most of the other exchanges in TLS 1.3 are highly optimized and do not require compression to be used.

The compression profile defining the use of algorithms, algorithm parameters, and extensions is specified via a JSON dictionary.

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"
}
```

The following elements are defined:

profile (integer): identifies the profile being defined.

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.

random (integer): indicates that the ClientHello.Random and ServerHello.Random values are truncated to the given length. 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. The Random length MUST be less than or equal to 32 bytes.

[[Open Issue: Karthik Bhargavan suggested the idea of hashing ephemeral public keys and to use the result (truncated to 32 bytes) as random values. Such a change would require a security analysis.]]

mutualAuth (boolean): if set to true, indicates that the client must authenticate with a certificate by sending Certificate and a CertificateVerify message. The server MUST omit the CertificateRequest message, as its contents are redundant. [[OPEN ISSUE: We don't actually say that you can omit empty messages, so we need to add that somewhere.]]

extension_order: indicates in what order extensions appear in respective messages. This allows to omit sending the type. If there is only a single extension to be transmitted, then the extension length field can also be omitted. For example, imagine that only the KeyShare extension needs to be sent in the ClientHello as the only extension. Then, the following structure

```
28                     // Extensions.length
33 26                  // KeyShare
  0024                 // client_shares.length
    001d               // KeyShareEntry.group
    0020 a690...af948 // KeyShareEntry.key_exchange
```

is compressed down to (assuming the KeyShare group has been pre-agreed)

```
0020 a690...af948 // KeyShareEntry.key_exchange
```

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.]]

2.1.1. Requirements on TLS Implementations

To be compatible with the specializations described in this section, a TLS stack needs to provide the following 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.

2.1.2. 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:

* 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.

* 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

* 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:

* 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.

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

* 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.]]

2.1.3. 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 certificates 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.

2.2. Record Layer

The only cTLS records that are sent in plaintext are handshake records (ClientHello and ServerHello/HRR). The content type is therefore constant (it is always handshake), so we instead set the content_type field to a fixed cTLS-specific value to distinguish cTLS plaintext records from encrypted records, TLS/DTLS records, and other protocols using the same 5-tuple.

The profile_id field allows the client and server to agree on which compression profile should be used for this session (see Section 2.1). This field MUST be set to zero if and only if no compression profile is used. Non-zero values are negotiated out of band between the client and server, as part of the specification of the compression profile.

```
   struct {
       ContentType content_type = ctls_handshake;
       opaque profile_id<0..2^8-1>;
       opaque fragment<0..V>;
   } CTLSPlaintext;
```

[[OPEN ISSUE: The profile_id is needed in the ClientHello to inform the server what compression profile to use. For a ServerHello this field is not required. Should we make this field optional?]]

Encrypted records use DTLS 1.3 record framing, comprising a configuration octet followed by optional connection ID, sequence number, and length fields.

```
   0 1 2 3 4 5 6 7
   +-+-+-+-+-+-+-+-+
   |0|0|1|C|S|L|E E|
   +-+-+-+-+-+-+-+-+
   | Connection ID |   Legend:
   | (if any,      |
   /  length as    /   C   - Connection ID (CID) present
   |  negotiated)  |   S   - Sequence number length
   +-+-+-+-+-+-+-+-+   L   - Length present
   | 8 or 16 bit   |   E   - Epoch
   |Sequence Number|
   | (if present)  |
   +-+-+-+-+-+-+-+-+
   | 16 bit Length |
   | (if present)  |
   +-+-+-+-+-+-+-+-+

   struct {
       opaque unified_hdr[variable];
       opaque encrypted_record[length];
   } CTLSCiphertext;
```

The presence and size of the connection ID field is negotiated as in DTLS.

As with DTLS, the length field MAY be omitted by clearing the L bit, which means that the record consumes the entire rest of the data in the lower level transport. In this case it is not possible to have multiple DTLSCiphertext format records without length fields in the same datagram. In stream-oriented transports (e.g., TCP), the length field MUST be present. For use over other transports length information may be inferred from the underlying layer.

Normal DTLS does not provide a mechanism for suppressing the sequence number field entirely. In cases where a sequence number is not required (e.g., when a reliable transport is in use), a cTLS implementation may suppress it by setting the suppressSequenceNumber flag in the compression profile being used (see Section 2.1). When this flag is enabled, the S bit in the configuration octet MUST be cleared.

2.3. Handshake Layer

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

* The length field is omitted.

* 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. 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.

3.1. ClientHello

The cTLS ClientHello is defined as follows.

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

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

3.2. ServerHello

We redefine ServerHello in the following way.

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

3.3. HelloRetryRequest

The HelloRetryRequest has the following format.

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

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

4. Examples

This section provides some example specializations.

For this example we use TLS 1.3 only with AES_GCM, X25519, ALPN h2, short random values, and everything else is ordinary TLS 1.3.

```
{
   "Version" : 0x0304
   "Profile" : 1,
   "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?]]

5. 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.

6. IANA Considerations

This document requests that a code point be allocated from the "TLS ContentType registry. This value must be in the range 0-31 (inclusive). The row to be added in the registry has the following form:

Value	Description	DTLS-OK	Reference
TBD	ctls	N	RFCXXXX

Table 1

[[RFC EDITOR: Please replace the value TBD with the value assigned by IANA, and the value XXXX to the RFC number assigned for this document.]]

[[OPEN ISSUE: Should we require standards action for all profile IDs that would fit in 2 octets.]]

7. 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. Example Exchange

The follow exchange illustrates a complete cTLS-based exchange supporting mutual authentication using certificates. The digital signatures use ECDSA with SHA256 and NIST P256r1. The ephemeral Diffie-Hellman uses the FX25519 curve and the exchange negotiates TLS-AES-128-CCM8-SHA256. The certificates are exchanged using certificate identifiers.

The resulting byte counts are as follows:

	ECDHE		
	TLS	CTLS	Overhead
ClientHello	132	36	4
ServerHello	90	36	4
ServerFlight	478	80	7
ClientFlight	458	80	7
Total	1158	232	22

The following compression profile was used in this example:

```
{
  "profile": 1,
  "version": 772,
  "cipherSuite": "TLS_AES_128_CCM_8_SHA256",
  "dhGroup": "X25519",
  "signatureAlgorithm": "ECDSA_P256_SHA256",
  "finishedSize": 8,
  "clientHelloExtensions": {
    "server_name": "000e00000b6578616d706c652e636f6d",
  },
  "certificateRequestExtensions": {
    "certificate_request_context": 0,
    "signature_algorithms": "00020403"
  },
  "mutualAuth": true,
  "extension-order": {
       "clientHelloExtensions": {
          Key_share
       },
       "ServerHelloExtensions": {
          Key_share
       },
  },

  "knownCertificates": {
    "61": "3082...",
    "62": "3082...",
    "63": "...",
    "64": "...",
    ...
  }
}

   ClientHello: 36 bytes = DH(32) + Overhead(4)

01                     // ClientHello
01                     // Profile ID
0020 a690...af948      // KeyShareEntry.key_exchange

   ServerHello: 36 = DH(32) + Overhead(4)

02                  // ServerHello
26                  // Extensions.length
0020 9fbc...0f49    // KeyShareEntry.key_exchange

   Server Flight: 80 = SIG(64) + MAC(8) + CERTID(1) + Overhead(7)

   The EncryptedExtensions, and the CertificateRequest messages are
   omitted because they are empty.
```

```
0b                  // Certificate
  03                //   CertificateList
    01              //     CertData.length
      61            //       CertData = 'a'

0f                  // CertificateVerify
  4064              //   Signature.length
       3045...10ce //   Signature

14                  // Finished
  bfc9d66715bb2b04 //   VerifyData
```

   Client Flight: 80 bytes = SIG(64) + MAC(8) + CERTID(1) + Overhead(7)

```
0b                  // Certificate
  03                //   CertificateList
    01              //     CertData.length
      62            //       CertData = 'b'


0f                  // CertificateVerify
  4064              //   Signature.length
       3045...f60e //   Signature

14                  // Finished
  35e9c34eec2c5dc1 //   VerifyData
```

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