Workgroup: TLS Working Group
Internet-Draft: draft-ietf-tls-ctls-06
Published: 9 July 2022
Intended Status: Standards Track
Expires: 10 January 2023
Authors: E. Rescorla (Mozilla), R. Barnes (Cisco), H. Tschofenig (Arm Limited), B. Schwartz (Google)

Compact TLS 1.3

Abstract

This document specifies a "compact" version of TLS and DTLS. It is logically isomorphic to ordinary TLS, 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 or DTLS, but it should eventually be possible for a single server port to offer cTLS alongside TLS or DTLS.

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 "compact" versions of TLS [RFC8446] and DTLS [RFC9147], respectively known as "Stream cTLS" and "Datagram cTLS". cTLS provides equivalent security and functionality to TLS and DTLS, but it is 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.

  OPEN ISSUE: Semi-static and point compression are never mentioned again.

For the common (EC)DHE handshake with pre-established certificates, Stream 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.

  TODO: Update these values.

cTLS supports the functionality of TLS and DTLS 1.3, and is forward-compatible to future versions of TLS and DTLS. cTLS itself is versioned by CTLSTemplate.version (currently zero).

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. Thus, each specialization produces a new protocol that preserves the security guarantees of TLS, but has its own unique handshake.

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 represented by the CTLSTemplate structure:

```
enum {
  profile(0),
  version(1),
  cipher_suite(2),
  dh_group(3),
  signature_algorithm(4),
  random(5),
  mutual_auth(6),
  handshake_framing(7),
  client_hello_extensions(8),
  server_hello_extensions(9),
  encrypted_extensions(10),
  cert_request_extensions(11),
  known_certificates(12),
  finished_size(13),
  optional(65535)
} CTLSTemplateElementType;

struct {
  CTLSTemplateElementType type;
  opaque data<0..2^32-1>;
} CTLSTemplateElement;

struct {
  uint16 ctls_version = 0;
  CTLSTemplateElement elements<0..2^32-1>
} CTLSTemplate;
```

TODO: Reorder enum.

Elements in a CTLSTemplate MUST appear sorted by the type field in strictly ascending order. The initial elements are defined in the subsections below. Future elements can be added via an IANA registry (Section 6.2). When generating a template, all elements are OPTIONAL to include. When processing a template, all elements are mandatory to understand (but see discussion of optional in Section 2.1.1.11).

For ease of configuration, an equivalent JSON dictionary format is also defined. It consists of a dictionary whose keys are the name of each element type (converted from snake_case to camelCase), and whose values are a type-specific representation of the element intended to maximize legibility. The cTLS version is represented by the key "ctlsVersion", whose value is an integer, defaulting to 0 if omitted.

OPEN ISSUE: Is it really worth converting snake_case to camelCase? camelCase is slightly more traditional in JSON, and saves one byte, but it seems annoying to implement.

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.

```
{
  "ctlsVersion": 0,
  "profile": "0001020304050607",
  "version": 772,
  "cipherSuite": "TLS_AES_128_GCM_SHA256"
}
```

2.1.1. Initial template elements

TODO: Reorder section.

2.1.1.1. profile

This element identifies the profile being defined. Its binary value is:

```
opaque ProfileID<1..2^8-1>
```

This encodes the profile ID, if one is specified. IDs whose decoded length is 4 bytes or less are reserved (see Section 6.5). When a reserved value is used (including the default value), other keys MUST NOT appear in the template, and a client MUST NOT accept the template unless it recognizes the ID.

In JSON, the profile ID is represented as a hexadecimal-encoded string.

2.1.1.2. version

Value: a single ProtocolVersion ([RFC8446], Section 4.1.2) that both parties agree to use. For TLS 1.3, the ProtocolVersion is 0x0304.

When this element is included, the supported_versions extension is omitted from ClientHello.extensions.

In JSON, the version is represented as an integer (772 = 0x0304 for TLS 1.3).

2.1.1.3. cipher_suite

Value: a single CipherSuite ([RFC8446], Section 4.1.2) that both parties agree to use.

When this element is included, the ClientHello.cipher_suites and ServerHello.cipher_suite fields are omitted.

In JSON, the cipher suite is represented using the "TLS_AEAD_HASH" syntax defined in [RFC8446], Section 8.4.

2.1.1.4. dh_group

Value: a single NamedGroup ([RFC8446], Section 4.2.7) to use for key establishment.

This is equivalent to a literal "supported_groups" extension consisting solely of this group.

Static vectors (see Section 2.1.2):

* KeyShareClientHello.client_shares

* KeyShareEntry.key_exchange, if the NamedGroup uses fixed-size key shares.

In JSON, the group is listed by the code point name in [RFC8446], Section 4.2.7 (e.g., "x25519").

2.1.1.5. signature_algorithm

Value: a single SignatureScheme ([RFC8446], Section 4.2.3) to use for authentication.

This is equivalent to a literal "signature_algorithms" extension consisting solely of this group.

In JSON, the signature algorithm is listed by the code point name in [RFC8446], Section 4.2.3. (e.g., ecdsa_secp256r1_sha256).

2.1.1.6. random

Value: a single uint8.

The ClientHello.Random and ServerHello.Random values are truncated to the given length. Where a 32-byte Random is required, 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.

In JSON, the length is represented as an integer.

2.1.1.7. mutual_auth

Value: a single uint8, with 1 representing "true" and 0 representing "false". All other values are forbidden.

If set to true, this element 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.

In JSON, this value is represented as true or false.

2.1.1.8. client_hello_extensions, server_hello_extensions, encrypted_extensions, and cert_request_extensions

Value: a single CTLSExtensionTemplate struct:

```
struct {
  Extension predefined_extensions<0..2^16-1>;
  ExtensionType expected_extensions<0..2^16-1>;
  uint8 allow_additional;
} CTLSExtensionTemplate;
```

The predefined_extensions field indicates extensions that should be treated as if they were included in the corresponding message. This allows these extensions to be omitted entirely.

The expected_extensions field indicates extensions that must be included in the corresponding message, at the beginning of its extensions field. The types of these extensions are omitted when serializing the extensions field of the corresponding message.

The allow_additional field MUST be 0 (false) or 1 (true), indicating whether additional extensions are allowed here.

predefined_extensions and expected_extensions MUST be in strictly ascending order by ExtensionType, and a single ExtensionType MUST NOT appear in both lists. If the version, dh_group, or signature_algorithm element appears in the template, the corresponding ExtensionType MUST NOT appear here.

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

Static vectors (see Section 2.1.2):

* Extension.extension_data for any extension in expected_extensions whose value has fixed length. This applies only to the corresponding message.

* The extensions field of the corresponding message, if allow_additional is false.

In JSON, this value is represented as a dictionary with three keys:

* predefinedExtensions: a dictionary mapping ExtensionType names ([RFC8446], Section 4.2) to values encoded as hexadecimal strings.

* expectedExtensions: an array of ExtensionType names.

* allowAdditional: true or false.

If predefinedExtensions or expectedExtensions is empty, it MAY be omitted.

> OPEN ISSUE: Should we have a certificate_entry_extensions element?

2.1.1.9. finished_size

Value: uint8, indicating that the Finished value is to be truncated to the given length.

> 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], Appendix E.1 for more on this, as well as https://mailarchive.ietf.org/arch/msg/tls/TugB5ddJu3nYg7chcyeIyUqWSbA.

In JSON, this length is represented as an integer.

2.1.1.10. handshake_framing

Value: uint8, with 0 indicating "false" and 1 indicating "true". If true, handshake messages MUST be conveyed inside a Handshake ([RFC8446], Section 4) struct on stream transports, or a DTLSHandshake ([RFC9147], Section 5.2) struct on datagram transports, and MAY be broken into multiple records as in TLS and DTLS. Otherwise, each handshake message is conveyed in a CTLSHandshake or CTLSDatagramHandshake struct (Section 2.3), which MUST be the payload of a single record.

In JSON, this value is represented as true or false.

2.1.1.11. optional

Value: a CTLSTemplate containing elements that are not required to be understood by the client. Server operators MUST NOT place an element in this section unless the server is able to determine whether the client is using it from the client data it receives. A key MUST NOT appear in both the main template and the optional section.

In JSON, this value is represented in the same way as the CTLSTemplate itself.

2.1.1.12. known_certificates

Value: a CertificateMap struct:

```
struct {
  opaque id<1..2^8-1>;
  opaque cert_data<1..2^16-1>;
} CertificateMapEntry;

struct {
  CertificateMapEntry entries<2..2^24-1>;
} CertificateMap;
```

Entries in the certificate map must appear in strictly ascending lexicographic order by ID.

In JSON, CertificateMap is represented as a dictionary from id to cert_data, which are both represented as hexademical strings:

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

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.

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.

2.1.2. Static vector compression

Some cTLS template elements imply that certain vectors (as defined in [RFC8446], Section 3.4) have a fixed number of elements during the handshake. These template elements note these "static vectors" in their definition. When encoding a "static vector", its length prefix is omitted.

For example, suppose that the cTLS template is:

```
{
  "ctlsVersion": 0,
  "version": 772,
  "dhGroup": "x25519",
  "clientHelloExtensions": {
    "expectedExtensions": ["key_share"],
    "allowAdditional": false
  }
}
```

Then, the following structure:

```
28                 // length(extensions)
33 26              // extension_type = KeyShare
  0024             // length(client_shares)
    001d           // KeyShareEntry.group
    0020           // length(KeyShareEntry.key_exchange)
      a690...af948 // KeyShareEntry.key_exchange
```

is compressed down to:

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

according to the following rationale:

* The length of extensions is omitted because allowAdditional is false, so the number of items in extensions (i.e., 1) is known in advance.

* extension_type is omitted because it is specified by expected_extensions.

* The length of client_shares is omitted because the use of dhGroup implies that there can only be one KeyShareEntry.

* KeyShareEntry.group is omitted because it is specified by dhGroup.

* The length of the key_exchange is omitted because the "x25519" key share has a fixed size (32 bytes).

2.2. Record Layer

The only cTLS records that are sent in plaintext are handshake records (ClientHello and ServerHello/HRR) and alerts. cTLS alerts are the same as TLS/DTLS alerts and use the same content types. For handshake records, we 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.

```
struct {
    ContentType content_type = ctls_handshake;
    opaque profile_id<0..2^8-1>;
    opaque fragment<0..2^16-1>;
} CTLSClientPlaintext;
```

The profile_id field MUST identify the profile that is in use. A zero-length ID corresponds to the cTLS default protocol. The server's reply does not include the profile_id, because the server must be using the same profile indicated by the client.

```
struct {
    ContentType content_type = ctls_handshake;
    opaque fragment<0..2^16-1>;
} CTLSServerPlaintext;
```

Encrypted records use DTLS 1.3 [RFC9147] record framing, comprising a configuration octet followed by optional connection ID, sequence number, and length fields. The encryption process and additional data are also as described in DTLS.

```
 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. When a reliable, ordered transport (e.g., TCP) is in use, the S bit in the configuration octet MUST be cleared and the sequence number MUST be omitted. When an unreliable transport is in use, the S bit has its usual meaning and the sequence number MUST be included.

2.3. cTLS Handshake Layer

The cTLS handshake is modeled in three layers:

1. The Transport layer

2. The Transcript layer

3. The Logical layer

2.3.1. The Transport layer

When template.handshake_framing is false, the cTLS transport layer uses a custom handshake framing that saves space by relying on the record layer for message lengths. (This saves 3 bytes per message compared to TLS, or 9 bytes compared to DTLS.) This compact framing is defined by the CTLSHandshake and CTLSDatagramHandshake structs.

Any handshake type registered in the IANA TLS HandshakeType Registry can be conveyed in a CTLS[Datagram]Handshake, but not all messages are actually allowed on a given connection. This definition shows the messages types supported in CTLSHandshake as of TLS 1.3 and DTLS 1.3, but any future message types are also permitted.

```
struct {
    HandshakeType msg_type;    /* handshake type */
    select (CTLSHandshake.msg_type) {
        case client_hello:          ClientHello;
        case server_hello:          ServerHello;
        case hello_retry_request:   HelloRetryRequest;  /* New */
        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;
        case request_connection_id: RequestConnectionId;
        case new_connection_id:     NewConnectionId;
    };
} CTLSHandshake;

struct {
    HandshakeType msg_type;    /* handshake type */
    uint16 message_seq;        /* DTLS-required field */
    select (CTLSDatagramHandshake.msg_type) {
      ... /* same as CTLSHandshake */
    };
} CTLSDatagramHandshake;
```

Each CTLSHandshake or CTLSDatagramHandshake MUST be conveyed as a single CTLSClientPlaintext.fragment, CTLSServerPlaintext.fragment, or CTLSCiphertext.encrypted_record, and is therefore limited to a maximum length of 2^16-1 or less. When operating over UDP, large CTLSDatagramHandshake messages will also require the use of IP fragmentation, which is sometimes undesirable. Operators can avoid these concerns by setting template.handshakeFraming = true.

2.3.2. The Transcript layer

TLS and DTLS start the handshake with an empty transcript. cTLS is different: it starts the transcript with a "virtual message" whose HandshakeType is ctls_template (Section 6.3) containing the CTLSTemplate used for this connection. This message is included in the transcript even though it is not exchanged during connection setup, in order to ensure that both parties are using the same template. Subsequent messages are appended to the transcript as usual.

When computing the handshake transcript, all handshake messages are represented in TLS Handshake messages, as in DTLS 1.3 ([RFC9147], Section 5.2), regardless of template.handshake_framing.

To ensure that all parties agree about what protocol is in use, the Cryptographic Label Prefix used for the handshake SHALL be "Sctls " for Stream cTLS and "Dctls " for Datagram cTLS. (This is similar to the prefix substitution in Section 5.9 of [RFC9147]).

2.3.3. The Logical layer

The logical handshake layer consists of handshake messages that are reconstructed following the instructions in the template. At this layer, predefined extensions are reintroduced, truncated Random values are extended, and all information is prepared to enable the cryptographic handshake and any import or export of key material and configuration.

There is no obligation to reconstruct logical handshake messages in any specific format, and client and server do not need to agree on the precise representation of these messages, so long as they agree on their logical contents.

3. Handshake Messages

In general, we retain the basic structure of each individual TLS or DTLS 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..2^16-1>;
    Extension extensions<1..2^16-1>;
} ClientHello;
```

3.2. ServerHello

We redefine ServerHello in the following way.

```
struct {
    Random random;
    CipherSuite cipher_suite;
    Extension extensions<1..2^16-1>;
} ServerHello;
```

3.3. HelloRetryRequest

In cTLS, the HelloRetryRequest message is a true handshake message instead of a specialization of ServerHello. The HelloRetryRequest has the following format.

```
struct {
    CipherSuite cipher_suite;
    Extension extensions<2..2^16-1>;
} HelloRetryRequest;
```

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

OPEN ISSUE: Does server_hello_extensions apply to HelloRetryRequest?

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.

```
{
   "ctlsVersion": 0,
   "profile": "0504030201",
   "version" : 772,
   "random": 16,
   "cipherSuite" : "TLS_AES_128_GCM_SHA256",
   "dhGroup": "x25519",
   "clientHelloExtensions": {
      "predefinedExtensions": {
          "application_layer_protocol_negotiation" : "030016832",
      },
      "allowAdditional": true
    }
}
```

Version 772 corresponds to the hex representation 0x0304 (i.e. 1.3).

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

6.1. Adding a ContentType

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	Y	RFCXXXX
TBD	ctls_handshake	Y	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.

6.2. Template Keys

This document requests that IANA open a new registry entitled "cTLS Template Keys", on the Transport Layer Security (TLS) Parameters page, with a "Specification Required" registration policy and the following initial contents:

Name	Value	Reference
profile	0	(This document)
version	1	(This document)
cipher_suite	2	(This document)
dh_group	3	(This document)
signature_algorithm	4	(This document)
random	5	(This document)
mutual_auth	6	(This document)
handshake_framing	7	(This document)
client_hello_extensions	8	(This document)
server_hello_extensions	9	(This document)
encrypted_extensions	10	(This document)
cert_request_extensions	11	(This document)
known_certificates	12	(This document)
finished_size	13	(This document)
optional	65535	(This document)

Table 2

6.3. Adding a cTLS Template message type

IANA is requested to add the following entry to the TLS HandshakeType registry.

*Value: TBD

*Description: ctls_template

*DTLS-OK: ??? Not clear what to put here.

*Reference: (This document)

*Comment: Virtual message used in cTLS.

6.4. Activating the HelloRetryRequest MessageType

This document requests that IANA change the name of entry 6 in the TLS HandshakeType Registry from "hello_retry_request_RESERVED" to "hello_retry_request", and set its Reference field to this document.

6.5. Reserved profiles

This document requests that IANA open a new registry entitled "Well-known cTLS Profile IDs", on the Transport Layer Security (TLS) Parameters page, with the following columns:

* ID value: A sequence of 1-4 octets.

* Template: A JSON object.

* Note: An explanation or reference.

The ID values of length 1 are subject to a "Standards Action" registry policy. Values of length 2 are subject to an "RFC Required" policy. Values of length 3 and 4 are subject to a "First Come First Served" policy. Values longer than 4 octets are not subject to registration and MUST NOT appear in this registry.

The initial registry contents are:

ID value	Template	Note
[0x00]	{"version": 772}	cTLS 1.3-only

Table 3

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/rfc/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/rfc/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/rfc/rfc8446>.

[RFC9147] Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", RFC 9147, DOI 10.17487/RFC9147, April 2022, <https://www.rfc-editor.org/rfc/rfc9147>.

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	69	2
ServerHello	90	64	2
ServerFlight	478	73	5
ClientFlight	458	73	5
Total	1158	279	14

The following compression profile was used in this example:

```
{
  "ctlsVersion": 0,
  "profile": "abcdef1234",
  "version": 772,
  "cipherSuite": "TLS_AES_128_CCM_8_SHA256",
  "dhGroup": "x25519",
  "signatureAlgorithm": "ecdsa_secp256r1_sha256",
  "finishedSize": 8,
  "clientHelloExtensions": {
    "predefinedExtensions": {
      "server_name": "000e00000b6578616d706c652e636f6d"
    },
    "expectedExtensions": ["key_share"],
    "allowAdditional": false
  },
  "serverHelloExtensions": {
    "expectedExtensions": ["key_share"],
    "allowAdditional": false
  },
  "certificateRequestExtensions": {
    "predefinedExtensions": {
      "certificate_request_context": "00",
      "signature_algorithms": "00020403"
    },
    "allowAdditional": false
  },
  "mutualAuth": true,
  "knownCertificates": {
    "61": "3082...",
    "62": "3082...",
    "63": "...",
    "64": "...",
    ...
  }
}
```


ClientHello: 71 bytes = Profile ID(5) + Random(32) + DH(32) + Overhead(2)

```
01                     // Handshake.msg_type = ClientHello
05 abcdef1234          // ClientHello.profile_id
5856a1...43168c130     // ClientHello.random
a690...af948           // KeyShareEntry.key_exchange
```

ServerHello: 65 bytes = Random(32) + DH(32) + Overhead(1)

```
02                       // Handshake.msg_type = ServerHello
cff4c0...684c859ca8      // ServerHello.random
9fbc...0f49              // KeyShareEntry.key_exchange
```

Server Flight: 78 = SIG(64) + MAC(8) + CERTID(1) + Overhead(5)

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

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

0f                   // CertificateVerify
  3045...10ce        //   signature

14                   // Finished
  bfc9d66715bb2b04 //   VerifyData
```

Client Flight: 78 bytes = SIG(64) + MAC(8) + CERTID(1) + Overhead(5)

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


0f                   // CertificateVerify
  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

Benjamin M. Schwartz
Google

Email: bemasc@google.com