TLS Working Group
Internet-Draft
Intended status: Informational
Expires: September 12, 2019

E. Rescorla
Mozilla

March 11, 2019

Compact TLS 1.3
draft-rescorla-tls-ctls-01

Abstract

This document specifies a "compact" version of TLS 1.3. It is isomorphic to TLS 1.3 but saves space by aggressive use of defaults and tighter encodings. CTLS is not interoperable with TLS 1.3, but it should eventually be possible for the server to distinguish TLS 1.3 and CTLS handshakes.

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 September 12, 2019.

Table of Contents

1. Introduction

DDISCLAIMER: 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 two basic techniques:

- Default values for common configurations, thus avoiding the need to take up space on the wire.

- More compact encodings, omitting unnecessary values.

For the common (EC)DHE handshake with (EC)DHE and pre-established public keys, CTLS achieves an overhead of [TODO] bytes over the minimum required by the cryptovariables.

Although isomorphic, CTLS implementations cannot interoperate with TLS 1.3 implementations because the packet formats are non-interoperable. It is probably possible to make a TLS 1.3 server switch-hit between CTLS and TLS 1.3 but this specification does not define how.

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.

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>;
```

[[OPEN ISSUE: Should we just re-encode this directly in CBOR?. That might be easier for people, but I ran out of time.]]

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. Thus, 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.

Overhead: 1 byte per record.

3.3. Handshake Layer

The CTLS handshake layer is the same as the TLS 1.3 handshake layer except that the length is a varint.

```
        struct {
            HandshakeType msg_type;    /* handshake type */
            varint length;             // CHANGED
            select (Handshake.msg_type) {
                case client_hello:          ClientHello;
                case server_hello:          ServerHello;
                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;
```

Overhead: 2 bytes per handshake message (min).

[OPEN ISSUE: This can be shrunk to 1 byte in some cases if we are willing to use a custom encoding. There are 11 handshake types, so we can use the first 4 bits for the type and then the bottom 4 bits for an encoding of the length, but we would have to offset that by 16 or so to be able to have a meaningful impact.]]

3.4. Extensions

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

```
    struct {
        ExtensionType 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 are slightly modified for space reduction.

4.1. ClientHello

The CTLS ClientHello is as follows.

```
uint8 ProtocolVersion;  // 1 byte
opaque Random[16];      // shortened
uint8 CipherSuite;      // 1 byte

struct {
    ProtocolVersion versions<0..255>;
    Random random;
    CipherSuite cipher_suites<1..V>;
    Extension extensions[remainder_of_message];
} ClientHello;
```

[[TODO: Define single-byte mappings of the cipher suites and protocol version.]]

The versions list from "supported_versions" has moved into ClientHello.versions with versions being one byte, but with the modern semantics of the client offering N versions and the server picking one.

In order to conserve space, the following extensions have default values which apply if they are not present:

- SignatureAlgorithms: ed25519
- SupportedGroups: the list of groups present in the KeyShare extension.
- Pre-Shared Key Exchange Modes: psk_dhe_ke
- Certificate Type: A new TBD value indicating a key index.

As a practical matter, the only extension needed is the KeyShare extension, as defined below.

Overhead: 8 bytes (min)

- Versions: 1 + # Versions
- CipherSuites: 1 + # Suites
- Key shares: 2 + 2 * # shares

4.1.1. KeyShare

The KeyShare extension is redefined as:

```
uint8 NamedGroup;
struct {
    NamedGroup group;
    opaque key_exchange<1..V>;
} KeyShareEntry;

struct {
    KeyShareEntry client_shares[length of extension];
} KeyShareClientHello;
```

[[TODO: Need a mapping for 8-bit group ids]]

4.2. ServerHello

We redefine ServerHello in a similar way:

```
struct {
    ProtocolVersion version;
    Random random;
    CipherSuite cipher_suite;
    Extension extensions[remainder_of_message];
} ServerHello;
```

The extensions have the same default values as in ClientHello, so as a practical matter only KeyShare is needed.

Overhead: 6 bytes

- Version: 1
- Cipher Suite: 1
- KeyShare: 4 bytes

4.2.1. KeyShare

```
struct {
    KeyShareEntry server_share;
} KeyShareServerHello;
```

[[OPEN ISSUE: We could save one byte here by removing the length of the key share and another byte by only allowing the client to send one key share (so group wasn't needed)..]]

[[TODO: Need to define a single-byte list of NamedGroups]].

4.2.2. PreSharedKeys

[[TODO]]

4.3. EncryptedExtensions

Unchanged.

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

4.4. CertificateRequest

This message removes the certificate_request_context and re-encodes the extensions.

```
struct {
    Extension extensions[remainder of message];
} 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 {
    CertificateEntry certificate_list[rest of extension];
} Certificate;
```

For a single certificate, this message will have a minumum of 2 bytes of overhead for the two length bytes.

[[OPEN ISSUE: What should the default type be?]]

4.5.1. Key IDs

WARNING: This is a new feature which has not seen any analysis and so may have real problems.

[[OPEN ISSUE: Do we want this at all?]]

It may also be possible to slim down the Certificate message further, by adding a KeyID-based mode, in which they keys were just a table index. This would redefines Certificate as:

```
struct {
    varint key_id;
} KeyIdCertificate;

struct {
      select (certiticate_type):
          case RawPublicKey, x509:
              CertificateEntry certificate_list<0..2^24-1>;

          case key_id:
              KeyIdCertificate;
      }
  } Certificate;
```

This allows the use of a short key id. Note that this is orthogonal to the rest of the changes.

IMPORTANT: You really want to include the certificate in the handshake transcript somehow, but this isn't specified for how.

4.5.2. CertificateVerify

Remove the signature algorithm and assume it's tied to the key. Note that this does not work for RSA keys, but if we just decide to be EC only, it works fine.

```
struct {
    opaque signature[rest of message];
} CertificateVerify;
```

4.5.3. Finished

Unchanged.

4.5.4. HelloRetryRequest

[[TODO]]

5. Handshake Size Calculations

5.1. ECDHE w/ Signatures

We compute the total flight size with X25519 and P-256 signatures, thus the keys are 32-bytes long and the signatures 64 bytes, with a cipher with an 8 byte auth tag, as in AEAD_AES_128_CCM_8. [Note: GCM should not be used with a shortened tag.] Overhead estimates marked with *** have been verified with Mint. Others are hand calculations and so may prove to be approximate.

5.1.1. Flight 1 (ClientHello) ***

- Random: 16
- KeyShare: 32
- Message Overhead: 8
- Handshake Overhead: 2
- Record Overhead: 1
- Total: 59

5.1.2. Flight 2 (ServerHello..Finished)

ServerHello ***

- Random: 16
- KeyShare: 32
- Message Overhead: 6
- Handshake Overhead: 2
- Total: 56

EncryptedExtensions ***

- Handshake Overhead: 2
- Total: 2

CertificateRequest ***

- Handshake Overhead: 2
- Total: 2

Certificate

- Certificate: X
- Length bytes: 2
- Handshake Overhead: 2
- Total: 4 + X

CertificateVerify

- Signature: 64
- Handshake Overhead: 2
- Total: 66

Finished

- MAC: 32
- Overhead: 2
- Total: 34

Record Overhead: 2 bytes (2 records) + 8 bytes (auth tag).

[[OPEN ISSUE: We'll actually need a length field for the ServerHello, to separate it from the ciphertext.]]

Total Size: 175 + X bytes.

5.1.3. Flight 3 (Client Certificate..Finished)

Certificate

- Certificate: X
- Length bytes: 2
- Handshake Overhead: 2

- Total: 4 + X

CertificateVerify

- Signature: 64
- Handshake Overhead: 2
- Total: 66

Finished

- MAC: 32
- Handshake Overhead: 2
- Total: 34

Record Overhead: 1 byte + 8 bytes (auth tag)

Total: 113 + X bytes

5.2. ECDHE w/ PSK

[TODO]

6. 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, modulo use of new features such as KeyID certificate messages.

One piece that is a new TLS 1.3 feature is the addition of the key_id, which definitely requires some analysis, especially as it looks like a potential source of identity misbinding. This is entirely separable from the rest of the specification.

[[OPEN ISSUE: One could imagine internally translating CTLS to TLS 1.3 so that the transcript, etc. were the same, but I doubt it's worth it, and then you might need to worry about cross-protocol attacks.]]

7. IANA Considerations

This document has no IANA actions.

8. 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>.

Acknowledgments

TODO acknowledge.

Author's Address

Eric Rescorla
Mozilla

Email: ekr@rtfm.com