Network Working Group
Internet-Draft
Intended status: Experimental
Expires: September 6, 2019

J. Schaad

March 5, 2019

TLS Handshake in CBOR
draft-schaad-ace-tls-cbor-handshake-00

Abstract

None

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 6, 2019.

Copyright Notice

Copyright (c) 2019 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. Summary 2
2. Handshake Protocol 4
3. Client Hello 5
4. Server Hello 5
5. Extensions 6
5.1. Supported Versions 7
5.2. Cookie 7
5.3. Signature Algorithms 7
5.4. Certificate Authorities 9
5.5. OID Filters 9
5.6. Post-Handshake Client Authentication 9
5.7. Supported Groups 9
5.8. Key Share 10
5.8.1. ECDHE Parameters 11
5.9. Pre-Shared Key Exchange Modes 11
5.10. Early Data Indication 12
5.11. Pre-Shared Key Extension 12
5.12. Other Extensions 13
6. Server Parameters 13
6.1. Encrypted Extensions 13
7. Certificate Request 14
8. Authentication Messages 14
8.1. Certificate 14
8.2. Certificate Verify 15
8.3. Finish 15
9. Record Protocol 15
9.1. Record Layer 15
9.2. Record Payload Protection 16
10. CBOR Slashed Version 17
11. Transport with CoAP 17
12. Informational 18
Appendix A. Sample Messages 18
A.1. Standard TLS for X25519 and Ed25519 18
A.2. Pre-shared key for authentication w/ ephemeral DH 20
A.3. Stripped TLS w/ Ed25519 22
A.4. Stripped TLS w/ PSK 24
Appendix B. Open ideas for ways to make things smaller 25
Appendix C. EDHOC issues that worry me 26
Author's Address 26

1. Summary

There are two measures that can be looked at when measuring how good the different options presented below are going to be.

- Total Number Of Bytes: The total number of bytes that are needed to perform the key establishment protocol.

- Total Number of Messages: The total number of messages that are needed to perform the key establishment protocol. This measure is going to be dependent on which of the transports is being used. For UDP the sizes are going to be from 576 to 1152 in the course of normal events. This means that all of the protocols being looked at can be run in three messages without any problems. For 6LoWPAN L2 however, the packet size is 127 bytes which leads to many more messages needed to be used. One can send approximately 80 bytes of payload in a single message, but if one is using blockwise transfer then the payloads are going to be limited to 64 bytes.

Summary of message sizes in bytes.

	TLS	TLS	TLS-C	TLS-C	TLS-S	TLS-S
	RPK	PSK	RPK	PSK	RPK	PSK
Message #1	122	164	97	134	47	64
Message #2	306	163	262	143	145	66
Message #3	205	72	175	54	101	21
Total	633	399	534	331	293	151

Summary of message sizes in estimated number of messages.

	TLS	TLS	TLS-C	TLS-C	TLS-S	TLS-S
	RPK	PSK	RPK	PSK	RPK	PSK
Message #1	2	3	2	3	1	1
Message #2	5	3	5	3	3	1
Message #3	4	1	3	1	2	1
Total	11	7	10	7	6	3

2. Handshake Protocol

The TLS Handshake message is:

```
struct {
    HandshakeType msg_type;    /* handshake type */
    uint24 length;             /* remaining bytes in message */
    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;
```

For the CBOR encoding make the following changes:

- Remove the length field: It is expected that this can be passed down from the record layer.

- Make the message type and field be a map.

```
Handshake = handshakeMessage

handshakeMessage = {
   1 : ClientHello,
   2 : ServerHello,
   3 : EndOfEarlyData,
   4 : EncryptedExtensions,
   5 : CertificateRequest,
   6 : Certificate,
   7 : CertificateVerify,
   8 : Finished,
   9 : NewSessionTicket,
   10 : KeyUpdate
}
```

Expected space savings:

- 3 bytes for the length

3. Client Hello

The TLS Client Hello structure is:

```
uint16 ProtocolVersion;
opaque Random[32];

uint8 CipherSuite[2];    /* Cryptographic suite selector */

struct {
    ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
    Random random;
    opaque legacy_session_id<0..32>;
    CipherSuite cipher_suites<2..2^16-2>;
    opaque legacy_compression_methods<1..2^8-1>;
    Extension extensions<8..2^16-1>;
} ClientHello;
```

For the CBOR encoding make the following changes:

- Remove all of the legacy items - reduction of 5 bytes + lengths?.

```
ClientHello = [
    random : bstr .len 32,
    cipher_suites : [+ int . len 1], // max 2^8-1
    extensions : [ + Extension ]
]
```

4. Server Hello

The TLS Server Hello structure is:

```
struct {
    ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
    Random random;
    opaque legacy_session_id_echo<0..32>;
    CipherSuite cipher_suite;
    uint8 legacy_compression_method = 0;
    Extension extensions<6..2^16-1>;
} ServerHello;
```

For the CBOR encoding make the following changes:

- Remove all of the legacy fields.

```
ServerHello = [
   random : bstr .len 32
   cipher_suite : int,
   extensions : [ + Extension ]
]
```

5. Extensions

The TLS Extenions field is:

```
struct {
     ExtensionType extension_type;
     opaque extension_data<0..2^16-1>;
 } Extension;

 enum {
     server_name(0),                             /* RFC 6066 */
     max_fragment_length(1),                     /* RFC 6066 */
     status_request(5),                          /* RFC 6066 */
     supported_groups(10),                       /* RFC 8422, 7919 */
     signature_algorithms(13),                   /* RFC 8446 */
     use_srtp(14),                               /* RFC 5764 */
     heartbeat(15),                              /* RFC 6520 */
     application_layer_protocol_negotiation(16), /* RFC 7301 */
     signed_certificate_timestamp(18),           /* RFC 6962 */
     client_certificate_type(19),                /* RFC 7250 */
     server_certificate_type(20),                /* RFC 7250 */
     padding(21),                                /* RFC 7685 */
     pre_shared_key(41),                         /* RFC 8446 */
     early_data(42),                             /* RFC 8446 */
     supported_versions(43),                     /* RFC 8446 */
     cookie(44),                                 /* RFC 8446 */
     psk_key_exchange_modes(45),                 /* RFC 8446 */
     certificate_authorities(47),                /* RFC 8446 */
     oid_filters(48),                            /* RFC 8446 */
     post_handshake_auth(49),                    /* RFC 8446 */
     signature_algorithms_cert(50),              /* RFC 8446 */
     key_share(51),                              /* RFC 8446 */
     (65535)
 } ExtensionType;
```

For the CBOR encoding make the following changes:

- Remove items we don't care about

- Encode as a map.

```
ExtensionType = (
//  0 : ServerName,
//  1 : max_fragment_length,
//  5 : status_request,
  10 : supported_groups,
  13 : signature_algorithms,
//  14 : use_srtp,
//  15 : heartbeat,
//  16 : application_layer_protocol_negoiation,
//  18 : signed_certificate_timestamp,
//  19 : client_certificate_type,
//  20 : server_certificate_type,
//  21 : padding,
  41 : pre_shared_key,
//  42 : early_data,
//  43 : supported_versions,
//  44 : cookie,
  45 : psk_key_exchange_modes,
//  47 : certificate_authorities,
//  48 : oid_filters,
//  49 : post_handshake_auth,
//  50 : signature_algoirthms_cert,
  51 : key_share
)

Extension = ( int, any )
```

5.1. Supported Versions

Not currently used as we only have one version. If absent it will be assumed to be this version.

5.2. Cookie

There is no reason for this extension to be used in CoRE. For the same functionality use [I-D.ietf-core-echo-request-tag].

5.3. Signature Algorithms

The TLS signature algorithm structures are:

```
   enum {
       /* RSASSA-PKCS1-v1_5 algorithms */
       rsa_pkcs1_sha256(0x0401),
       rsa_pkcs1_sha384(0x0501),
       rsa_pkcs1_sha512(0x0601),

       /* ECDSA algorithms */
       ecdsa_secp256r1_sha256(0x0403),
       ecdsa_secp384r1_sha384(0x0503),
       ecdsa_secp521r1_sha512(0x0603),

       /* RSASSA-PSS algorithms with public key OID rsaEncryption */
       rsa_pss_rsae_sha256(0x0804),
       rsa_pss_rsae_sha384(0x0805),
       rsa_pss_rsae_sha512(0x0806),

       /* EdDSA algorithms */
       ed25519(0x0807),
       ed448(0x0808),

       /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */
       rsa_pss_pss_sha256(0x0809),
       rsa_pss_pss_sha384(0x080a),
       rsa_pss_pss_sha512(0x080b),

       /* Legacy algorithms */
       rsa_pkcs1_sha1(0x0201),
       ecdsa_sha1(0x0203),

       /* Reserved Code Points */
       private_use(0xFE00..0xFFFF),
       (0xFFFF)
   } SignatureScheme;

   struct {
       SignatureScheme supported_signature_algorithms<2..2^16-2>;
   } SignatureSchemeList;
```

One of the differences that may need to be dealt with at this point is the question of keeping the same enumeration as TLS uses or if the enumeration should be changed. For this document the same enumeration is being kept. TLS uses the current two byte format because it separates the hash algorithm from the public key algorithms. For a single algorithm this ends up being 3 bytes for CBOR and 4 bytes for TLS. Each additional algorithm adds 2 bytes until you get to 12 algorithms. If one switched to using integer values from the COSE tables, then one ends up with the same byte count.

For the CBOR encoding make the following changes:

o None

```
signature_algorithms = bstr
```

5.4. Certificate Authorities

Not used.

5.5. OID Filters

Not used.

5.6. Post-Handshake Client Authentication

Not used.

5.7. Supported Groups

The TLS structure is:

```
enum {

    /* Elliptic Curve Groups (ECDHE) */
    secp256r1(0x0017), secp384r1(0x0018), secp521r1(0x0019),
    x25519(0x001D), x448(0x001E),

    /* Finite Field Groups (DHE) */
    ffdhe2048(0x0100), ffdhe3072(0x0101), ffdhe4096(0x0102),
    ffdhe6144(0x0103), ffdhe8192(0x0104),

    /* Reserved Code Points */
    ffdhe_private_use(0x01FC..0x01FF),
    ecdhe_private_use(0xFE00..0xFEFF),
    (0xFFFF)
} NamedGroup;

struct {
    NamedGroup named_group_list<2..2^16-1>;
} NamedGroupList;
```

It makes more sense to change the enumeration from that used by TLS to the COSE EC curve registry as those values are only single byte values and are small. One implication is that all of the Finite Field Groups are dropped, but this should not be a problem. This means a 2 byte value for a single curve in the CBOR version rather

than a 4 byte encoding for TLS. Adding a second curve adds one byte for CBOR and 2 bytes for TLS.

For the CBOR encoding make the following changes:

- Change from a two byte enumeration to the the integer values from the COSE EC curve registry.

```
NamedGroup = {
    secp256r1: 1, secp384r1: 2, secp521r1: 3,
    x25519: 4, x448:5
}

supported_groups = [ + NamedGroup ]
```

5.8. Key Share

The TLS structure for key share is:

```
struct {
    NamedGroup group;
    opaque key_exchange<1..2^16-1>;
} KeyShareEntry;

struct {
    KeyShareEntry client_shares<0..2^16-1>;
} KeyShareClientHello;

struct {
    NamedGroup selected_group;
} KeyShareHelloRetryRequest;

struct {
   KeyShareEntry server_share;
} KeyShareServerHello;
```

For the CBOR encoding make the following changes:

```
keyShareEntry = {
    secp256r1 : CompressedPointRepresentation,
    secp384r1 : CompressedPointRepresentation,
    secp521r1 : CompressedPointRepresentation,
    x25519 : bstr,
    x448 : bstr,
    * NamedGroup : any
}

key_share = KeyShare_ClientHello | KeyShare_HelloRetryRequest |
        KeyShare_ServerHello

KeyShare_ClientHello = [ *keyShareEntry]
KeyShare_HelloRetryRequest = NamedGroup
KeyShare_ServerHello = keyShareEntry
```

5.8.1. ECDHE Parameters

The TLS structure is:

```
struct {
    uint8 legacy_form = 4;
    opaque X[coordinate_length];
    opaque Y[coordinate_length];
} UncompressedPointRepresentation;
```

For the CBOR encoding make the following changes:

o Switched to compressed points for space savings.

```
CompressedPointReprentation = [
    x : bstr,
    y : bool
]
```

5.9. Pre-Shared Key Exchange Modes

The TLS structure is:

```
enum { psk_ke(0), psk_dhe_ke(1), (255) } PskKeyExchangeMode;

struct {
    PskKeyExchangeMode ke_modes<1..255>;
} PskKeyExchangeModes;
```

Changes for CBOR:

```
pskKeyExchangeMode = ( psk_key: 0, psk_dhe_ke:1 }

psk_key_exchange_modes = [ + pskKeyExchangeMode]
```

5.10. Early Data Indication

Not used.

5.11. Pre-Shared Key Extension

The TLS structure is:

```
struct {
    opaque identity<1..2^16-1>;
    uint32 obfuscated_ticket_age;
} PskIdentity;

opaque PskBinderEntry<32..255>;

struct {
    PskIdentity identities<7..2^16-1>;
    PskBinderEntry binders<33..2^16-1>;
} OfferedPsks;

struct {
    select (Handshake.msg_type) {
        case client_hello: OfferedPsks;
        case server_hello: uint16 selected_identity;
    };
} PreSharedKeyExtension;
```

The changes for CBOR are:

```
pre_shared_key = clientHello_PSK | serverHello_PSK

clientHello_PSK = OfferedPsks

serverHello_PSK = int

OfferedPsks = [
   identities : [ +PskIdentity ],
   binders : bstr
]

PskIdentity = (
    identity : bstr,
    ? obfuscated_ticket_age : int
)
```

5.12. Other Extensions

```
TLS

  struct {
    select(ClientOrServerExtension) {
        case client:
            CertificateType client_certificate_types<1..2^8-1>;
        case server:
            CertificateType client_certificate_type;
   }
} ClientCertTypeExtension;

struct {
    select(ClientOrServerExtension) {
        case client:
            CertificateType server_certificate_types<1..2^8-1>;
        case server:
            CertificateType server_certificate_type;
    }
} ServerCertTypeExtension;

CBOR

clientCertType = certTypeRequest | certTypeResponse
serverCertType = certTypeRequest | certTypeResponse

certTypeRequest = [+ cerType]
certTypeResponse = certType

certType = (x509:0, rawPublicKey:1 }
```

6. Server Parameters

6.1. Encrypted Extensions

The TLS structure is:

```
struct {
    Extension extensions<0..2^16-1>;
} EncryptedExtensions;
```

For CBOR:

```
EncryptedExtensions = [ * Extension ]
```

7. Certificate Request

Not Used

8. Authentication Messages

8.1. Certificate

TLS

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

struct {
    select (certificate_type) {
        case RawPublicKey:
          /* From RFC 7250 ASN.1_subjectPublicKeyInfo */
          opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;

        case X509:
          opaque cert_data<1..2^24-1>;
    };
    Extension extensions<0..2^16-1>;
} CertificateEntry;

struct {
    opaque certificate_request_context<0..2^8-1>;
    CertificateEntry certificate_list<0..2^24-1>;
} Certificate;

certificate = [
    ? certificate_request_context : bstr,
    certificate_list : [* CertificateEntry]
]

CertificateEntry = [
    certificate : {
        0 : bstr, // cert_data,
        1 : bstr  // ASN1_subjectPublicKeyInfo
    },
    extensions : [* Extension]
]
```

8.2. Certificate Verify

TLS

```
struct {
    SignatureScheme algorithm;
    opaque signature<0..2^16-1>;
} CertificateVerify;
```

CBOR

```
CertificateVerify = [
    algorithm : SignatureScheme,
    signature : bstr
]
```

8.3. Finish

TLS

```
struct {
    opaque verify_data[Hash.length];
} Finished;
```

CBOR

```
Finished = bstr
```

9. Record Protocol

9.1. Record Layer

TLS

```
enum {
    invalid(0),
    change_cipher_spec(20),
    alert(21),
    handshake(22),
    application_data(23),
    (255)
} ContentType;

struct {
    ContentType type;
    ProtocolVersion legacy_record_version;
    uint16 length;
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;
```

CBOR

```
contentType = (
    invalid: 0, change_cipher_spec: 20, alert:21, handshake:22,
    application_data:23
)

TLSPlaintext = (
    type : contentType,
    fragment : bstr
)
```

9.2. Record Payload Protection

TLS

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

struct {
    ContentType opaque_type = application_data; /* 23 */
    ProtocolVersion legacy_record_version = 0x0303; /* TLS v1.2 */
    uint16 length;
    opaque encrypted_record[TLSCiphertext.length];
} TLSCiphertext;
```

CBOR

```
TLSCiphertext = (
    type : application_data,
    encrypted_record : bstr
)
```

10. CBOR Slashed Version

There are many things that the EDHOC system did that slashed down size that has not been done for the previous version of TLS. If these changes are made then a non-trivial amount of savings can be done that might or might not be considered acceptable in this situation.

For the puspose of making things even small we are making the following assumptions:

- Remove the random number strings from the system. Since we are always going to do new ephemeral keys on both sides when the protocol is done, then the random numbers are not needed. If new ephemeral keys are not done on both sides, then the perfect forward secrecy requirement is not met.

- Trucate all HMAC values to 64-bits.

- If the encrypted extensions handshake message is empty, then omit it.

- In cases where a single value can be in an array, change the syntax to be either the single value or an array of values.

- Change the behavior of some of the extensions by adding additional operations and changing defaults.

  * Change the default value for client and server certificate type from "certificate" to "RPK" and "reference".

  * Change the behavior of the curves extension so that if it is omitted, then the cruve offered in the key share is assumed to be the only one supported by the client.

11. Transport with CoAP

Transporting the messages w/ CoAP is fairly simple:

1. Message #1 is a POST to a fixed location.

2. Message #2 is the response to the POST. The message returns either Created (2.01) or a TLS Alert message with Bad Request

(4.00). If the message is successful, then a Location-Path will be returned as part of the message.

3. Message #3 is a PUT to the returned location path from the above response.

4. Message #4 is either a TLS Alert message with Bad Request (4.00) or an empty Changed (2.04) message. At this point the location path is deleted on the server.

12. Informational

[I-D.ietf-core-echo-request-tag]
Amsuess, C., Mattsson, J., and G. Selander, "Echo and Request-Tag", draft-ietf-core-echo-request-tag-03 (work in progress), October 2018.

Appendix A. Sample Messages

A.1. Standard TLS for X25519 and Ed25519

The size of messge #1 is 97 bytes

```
22,
<< 1,
   [
   h'00112233445566778899001122334455667788990011223344556
   667788990011',  / random /
      [ h'1304' / TLS_AES_128_CCM_SHA256 ], / cipher suites /
      [ 51, [ 4 / x25519 /, h'00112233445566778899001122334455
      667788990011223344556677889900 11' ],
        13, [ h'0807' / Ed2215 / ],  / signature algorithms /
        10, [ 4 / secp256r1 / ],
        19, [ 1 ] / client_cert_type /,
        20, [ 1 ] / server_cert_type /,
      ]
   ]
>>
```

Message #1

The size of message #2 is 262 bytes. If you directly encode what is below it will be 16 bytes short as there is no provision in the CDDL for the 16 bytes of the MAC appended to the end of the encrypted data.

```
   22,
   <<
      2,
      [
      h'00112233445566778899001122334455667788990011223344
      55667788990011',  / random /
         h'1304', / cipher suite /
         [
         51, [ 4 / x25519 /,  h'00112233445566778899001122334
         5566778899001122334455667788990011' ] / key share /
         ]
      ]
   >>,

   23,
   <<
      4,
      [ 19, 1, / client_cert_type /
        20, 1  / server_cert_type /
      ],
      11,
      [
         [ 1 / rpk /,
         h'11223344556677889900112233445566778899001122334455
         66778899001122334455667788990011223344'
         ]
      ],
      12,
      [
         h'0807',
         h'11223344556677889900112233445566778899001122334455
         66778899001122334455667788990011223344556677889900112233
         4455667788990011223344'
      ],
      13,
      h'11223344556677889900112233445566778899001122334455
      66778899001122'
   >>
```

Message #2

Size of message #3 is 175 bytes

```
   23,
   <<
      11,
      [
         [ 1 / rpk /,
         h'1122334455667788990011223344556677889900112233445
         667788990011223344556677889900112233 44'
         ]
      ],
      12,
      [
         h'0807',
         h'1122334455667788990011223344556677889900112233 44
         556677889900112233445566778899001122334455667788 99
         00112233445566778899001122334 4'
      ],
      13,
      h'1122334455667788990011223344556677889900112233445566
      778899001122'
   >>
```

Message #3

A.2. Pre-shared key for authentication w/ ephemeral DH

The size of messge #1 is 97 bytes

```
   22,
   << 1,
      [
      h'00112233445566778899001122334455667788
      9001122334455667788990011', / random /
         [ h'1304' / TLS_AES_128_CCM_SHA256 ], / cipher suites /
         [ 51, [ 4 / x25519 /, h'00112233445566778899001122334455
         5667788990011223344556677889900112233445' ],
           13, [ h'0807' / Ed2215 / ], / signature algorithms /
           10, [ 4 / secp256r1 / ],
           19, [ 1 ] / client_cert_type /,
           20, [ 1 ] / server_cert_type /,
         ]
      ]
   >>
```

Message #1

The size of message #2 is 262 bytes. If you directly encode what is below it will be 16 bytes short as there is no provision in the CDDL

```
   for the 16 bytes of the MAC appended to the end of the encrypted
   data.

   22,
   <<
      2,
      [
      h'00112233445566778899001122334455667788990011223344
      5667788990011',  / random /
         h'1304', / cipher suite /
         [
         51, [ 4 / x25519 /,  h'00112233445566778899001122
         33445566778899001122334455667788990011' ] / key share /
         ]
      ]
   >>,

   23,
   <<
      4,
      [ 19, 1, / client_cert_type /
        20, 1  / server_cert_type /
      ],
      11,
      [
         [ 1 / rpk /,
         h'11223344556677889900112233445566778899001122334455
         66778899001122334455667788990011223344'
         ]
      ],
      12,
      [
         h'0807',
         h'11223344556677889900112233445566778899001122334455
         66778899001122334455667788990011223344556678899001122
         3344556677889900112233 44'
      ],
      13,
      h'11223344556677889900112233445566778899001122334455667
        8899001122'
   >>
```

Message #2

Size of message #3 is 175 bytes

```
   23,
   <<
      11,
      [
         [ 1 / rpk /,
           h'11223344556677889900112233445566778899001122334455
             667788990011223344556677889900112233 44'
         ]
      ],
      12,
      [
         h'0807',
         h'11223344556677889900112233445566778899001122334455 66
          7788990011223344556677889900112233445566778899001122 33
          34455667788990011223344'
      ],
      13,
      h'11223344556677889900112233445566778899001122334455667 78
        899001122'
   >>
```

Message #3

A.3. Stripped TLS w/ Ed25519

The size of messge #1 is 47 bytes

```
   22,
   << 1,
      [
         1 / TLS_AES_128_CCM_SHA256_64 /,     / cipher suites /
         [ 1, [ 4 / x25519 /, h'00112233445566778899001122334455
                              667788990011223344556677889900 11' ],
           2, 99 / Ed2215 /                   / signature algorithms /
         ]
      ]
   >>
```

Message #1

The size of message #2 is 146 bytes. The encryption authentication code is added as a separate at the end of the encrypted handshake block.

```
   22,
   <<
      2,
      [
         1 / TLS_AES_128_CCM_SHA256_64 /,           / cipher suite /
         [
         1, [ 4 / x25519 /,  h'00112233445566778899011223344
               55667788990011223344556677889900 11' ] / key share /
         ]
      ]
   >>,

   23,
   <<
      11,
      [
         [ 9 / reference /,
           h'1122334455'
         ]  / certificate /
      ],
      12,
      [
         99,
         h'11223344556677889900112233445566778899001122
         33445566778899001122334455667788990011223344 55
         667788990011223344556677889900 11223344'
         / signature /
      ], / certificate verify /
      13,
      h'1122334455667788', / finish /
      h'11223344556677'  / encryption authentication code /
   >>
```

Message #2

Size of message #3 is 102 bytes

```
   23,
   <<
      11,
      [
         [ 9 / reference /,
           h'1122334455'
         ]
      ],  / certificate /
      12,
      [
         99,
         h'1122334455667788990011223344556678899001122
           3344556677889900112233445566778899001122334 4
           5566778899001122334455667788990011223344'
      ], / certificate verify /
      13,
      h'1122334455667788', / finish /
      h'1122334455667788'  / encryption authentication code /
   >>
```

Message #3

A.4. Stripped TLS w/ PSK

The size of messge #1 is 65 bytes

```
   22,
   << 1,
      [
         1,                   / cipher suites /
         [ 1, [ 4 / x25519 /, h'00112233445566778899001
                       22334455667788990011223344556677889900112233445566778899001122' ],
           2, 99,          / signature algorithms /
           6, [ h'0102030405', h'0102030405060708' ]
         ]
      ]
   >>
```

Message #1

The size of message #2 is 59 bytes. If you directly encode what is below it will be 16 bytes short as there is no provision in the CDDL for the 16 bytes of the MAC appended to the end of the encrypted data.

```
   22,
   <<
      2,
      [
         1, / cipher suite /
         [
           1, [ 4 / x25519 /,  h'00112233445566778899001122334455
                667788990011223344556677889900112233445566778899001122334455667788990011' ], / key share /
           6, 1
         ]
     ]
   >>,

   23,
   <<
      13,
      h'1122334455667788'
   >>
```

Message #2

Size of message #3 is 12 bytes

```
   23,
   <<
      13,
      h'1122334455667788'
   >>
```

Message #3

Appendix B. Open ideas for ways to make things smaller

The following things can still be considered for shrinking things:

- Make the random values optional. This would kill 66 bytes from the first and second messages.

- Think of a way to shorten the SKI encoding of the raw public key.

- Decode to re-encode the signature and cipher suite numbers

- Change some of the default values. For example we could change to RPK being the default certificate type.

- Create a HMAC cipher suite which truncates the output for doing finish messages. This is not unusual for the world of IoT and

thus should be considered. Halving the length of this saves 17 bytes in message 2 and 3.

Appendix C. EDHOC issues that worry me

I still need to actually read the current document. From a quick glance through I have the following issues:

- It is limited to four cipher suites of which only two are globally defined.

- Need to validate that P-256 can do key agreement correctly with only an X coordinate.

Author's Address

Jim Schaad

Email: ietf@augustcellars.com