LURK
Internet-Draft
Intended status: Standards Track
Expires: July 29, 2021

D. Migault
Ericsson

January 25, 2021

LURK Extension version 1 for (D)TLS 1.3 Authentication
draft-mglt-lurk-tls13-04

Abstract

This document describes the LURK Extension 'tls13' which enables interactions between a LURK Client and a LURK Server in a context of authentication with (D)TLS 1.3.

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 July 29, 2021.

Table of Contents

1. TODO

1. When information is missing in the handshake, LURK requires the length to be set to the appropriated format. This ease the use of a parser. TLS1.3 seems to consider the length as of the value of the expected field.

4.2.11.2. PSK Binder
[...]
The length fields for the message (including
the overall length, the length of the extensions block, and the
length of the "pre_shared_key" extension) are all set as if binders
of the correct lengths were present.

2. Introduction

This document defines a LURK extension for TLS 1.3 [RFC8446].

This document assumes the reader is familiar with TLS 1.3 the LURK architecture [I-D.mglt-lurk-lurk].

Interactions with the Cryptographic Service (CS) can be performed by the TLS Client as well as by the TLS Server.

LURK defines an interface to a CS that stores the security credentials which include the PSK involved in a PSK or PSK-ECDHE authentication or the key used for signing in an ECDHE authentication. In the case of session resumption the PSK is derived from the resumption_master_secret during the key schedule [RFC8446] section 7.1, this secret MAY require similar protection or MAY be delegated as in the LURK extension of TLS 1.2 [I-D.mglt-lurk-tls12].

The current document extends the scope of the LURK extension for TLS 1.2 in that it defines the CS on the TLS server as well as on the TLS client and the CS can operate in non delegating scenarios.

3. Terminology

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.

This document uses the terms defined [RFC8446] and [I-D.mglt-lurk-tls12].

4. LURK Header

LURK / TLS 1.3 is a LURK Extension that introduces a new designation "tls13". This document assumes that Extension is defined with designation set to "tls13" and version set to 1. The LURK Extension extends the LURKHeader structure defined in [I-D.mglt-lurk-lurk] as follows:

```
enum {
    tls13 (2), (255)
} Designation;

enum {
   capabilities(0),
   ping(1),
   s_init_cert_verify(2),
   s_new_ticket(3),
   s_init_early_secret(4),
   s_hand_and_app_secret(5),
   c_binder_key(6),
   c_init_early_secret(7),
   c_init_hand_secret(8),
   c_hand_secret(9),
   c_app_secret(10),
   c_cert_verify(11),
```

```
      c_register_ticket(12),
      c_post_hand(13), (255)
   }TLS13Type;


   enum {
       // generic values reserved or aligned with the
       // LURK Protocol
       request (0), success (1), undefined_error (2),
       invalid_payload_format (3),

       invalid_psk
       invalid_freshness

       invalid_request
       invalid_key_id_type
       invalid_key_id
       invalid_signature_scheme
       invalid_certificate_type
       invalid_certificate
       invalid_certificate_verify
       invalid_secret_request
       invalid_handshake
       invalid_extension
       invalid_ephemeral
       invalid_idnetity
       too_many_identities

   }TLS13Status

   struct {
        Designation designation = "tls13";
        int8 version = 1;
   } Extension;

   struct {
       Extension extension;
       select( Extension ){
           case ("tls13", 1):
               TLS13Type;
       } type;
       select( Extension ){
           case ("tls13", 1):
               TLS13Status;
       } status;
       uint64 id;
       unint32 length;
   } LURKHeader;
```

5. Overview

The CS is not expected to perform any policies such as choosing the appropriated authentication method. These are performed by the TLS client or TLS server that instruct the LURK client accordingly.

On the other hand, some CS MAY be optimized by implementing a subset of the specified possibilities described in this document. Typically some implementations MAY not implement the session resumption or the post handshake authentication to avoid keeping states of a given session once the handshake has been performed. These capabilities of the CS MAY also in return impact the policies of the TLS client or TLS server.

These limitations are mentioned throughout the document, and even represented in the state diagrams, the recommendation is that the CS SHOULD NOT impact the policies of the TLS client or TLS server. Instead they SHOULD be able to optimize the CS to their policies via some configuration parameters presented in section Section 12.1. Such parameters are implementation dependent and only provided here as informative.

This document defines the role to specify whether the CS runs on a TLS client or a TLS service. The CS MUST be associated a single role.

From a LURK client perspective, the purpose of the LURK exchange is to request secrets, a signing operations, or ticket (NewSessionTicket) as summed up in Table Figure 1.

Role	LURK exchange	secret	sign	ticket
server	s_init_early_secret	yes	-	-
server	s_init_cert_verify	yes	yes	-
server	s_hand_and_app_secret	yes	-	-
server	s_new_ticket	yes	-	yes
client	c_binder_key	yes	-	-
client	c_init_early_secret	yes	-	-
client	c_init_hand_secret	yes	-	-
client	c_hand_secret	yes	-	-
client	c_app_secret	yes	-	-
client	c_cert_verify	yes	yes	-
client	c_register_ticket	yes	-	yes
client	c_post_hand	-	yes	-

Figure 1: Operation associated to LURK exchange

The number of operations are limited, but the generation of secrets, tickets as well as signing heavily rely on the knowledge of the TLS handshake messages and in turn impacts these TLS handshake messages. As a result, these operations are highly inter-dependent. This is one reason multiple sequential exchanges are needed between the LURK client and the CS as opposed to independent requests for secrets, signing or tickets. This especially requires the necessity to create a session between the LURK client and the CS. In addition, the LURK client and the CS need to synchronize the TLS handshake. First it is a necessary component for the CS to generate the secrets, signature and tickets. Second, elements are respectively generated by the LURK client and by the CS.

While all these messages do share a lot of structures, they also require different structure that make them unique.

6. Structures

This section describes structures that are widely re-used across the multiple LURK exchanges.

6.1. secret_request

secret_request is a 16 bit structure described in Table Figure 2 that indicates the requested key or secrets by the LURK client. The secret_request structure is present in the request of any exchange except for a c_post_hand exchange. The same structure is used across all LURK exchanges, but each LURK exchange only permit a subset of values described in Table Figure 3.

A LURK client MUST NOT set secret_request to key or secrets that are not permitted. The CS MUST check the secret_request has only permitted values and has all mandatory keys or secrets set. If these two criteria are not met the CS MUST NOT perform the LURK exchange and SHOULD return a invalid_secret_request error. If the CS is not able to compute an optional key or secret, the CS MUST proceed the LURK exchange and ignore the optional key or secret.

Bit	key or secret (designation)
0	binder_key (b)
1	client_early_traffic_secret (e_c)
2	early_exporter_master_secret (e_x)
3	client_handshake_traffic_secret (h_c)
4	server_handshake_traffic_secret (h_s)
5	client_application_traffic_secret_0 (a_c)
6	server_application_traffic_secret_0 (a_s)
7	exporter_master_secret (x)
8	resumption_master_secret (r)
9-15	reserved and set to zero

Figure 2: secret_request structure

LURK exchange	Permitted secrets
s_init_cert_verify	h_c*, h_s*, a_c*, a_s*, x*
s_init_early_secret	b,e_c*, e_x*
s_hand_and_app_secret	h_c, h_s, a_c*, a_s*, x*
s_new_ticket	r*
c_binder_key	b
c_init_early_secret	e_c*, e_x*
c_init_hand_secret	h_c, h_s
c_hand_secret	h_c, h_s
c_app_secret	a_c*, a_s*, x*
c_cert_verify	a_c*, a_s*, x*
c_register_ticket	r*
c_post_hand	

Figure 3: secret_request permitted values per LURK exchange

6.2. handshake

The derivation of the secrets, signing operation and tickets requires the TLS handshake. The TLS handshake is described in [RFC8446] section 4 and maintained by the TLS server and the TLS client to derive the same secrets. As the CS is in charge is deriving the secrets as well to perform some signature verification, the CS must be be aware of the TLS handshake. The TLS handshake is not necessarily being provided by the LURK client to the CS, but instead is derived some structures provided by the LURK client as well as other structures generated or modified by the CS.

When an unexpected handshake context is received, the CS SHOULD return an invalid_handshake error.

The value of the TLS hanshake is defined in [RFC8446] section 4 and remainded in Table Figure 4 reminds the TLS handshake values after each LURK exchange and describes operations performed by the CS in order to build it.

On the TLS server:

- (a) ServerHello.random value provided by the LURK client requires specific treatment as described in Section 6.4 before being inserted in teh TLS handshake variable.

- (b) When the shared secret (and so the private ECDHE) is generated by the CS, the KeyShareServerHello structure cannot be provided to the CS by the LURK client in a ServerHello and is instead completed by the CS as described in Section 6.5.

- (c) The TLS server Certificate structure is not provided by the LURK client as part of the handshake structure. Instead, the CS generates the Certificate message from the certificate structure described in Section 6.7. The handshake MUST NOT contain a TLS Certificate message and CS SHOULD reject a handshake that contains a TLS Certificate message.

- (d) The Certificate and Finished messages are not provided in a handshake structure by the LURK client but are instead generated by the CS as described in Section 6.10.

TO FINALIZE THE TLS CLIENT

On the TLS client:

For s_init_cert_verify (resp. c_init_hand_secret) see Section 6.5 that describes how the KeyShareServerHello (resp. KeyShareClientHello) structure MAY be affected when the share secret is generated by the CS.

(e) ClientHello.random value provided by the LURK client requires specific treatment as described in Section 6.4 before being inserted in the TLS handshake variable. (f) When the shared secret (and so the private ECDHE) is generated by the CS, the KeyShareClientHello structure cannot be provided to the CS by the LURK client in a ServerHello and is instead completed by the CS as described in Section 6.5. (f)

Typically, shared secret MAY be generated by the CS (see Section 6.5) in which case, the public part that is part of the TLS handshake

is or signatures (see Section 6.10 are generated by the CS. structures that represent certificates (see Section 6.7) are provided in a separate message as to enable compression. In some cases, such as for s_init_cert_verify and c_cert_verify CertificateVerify and Finished messages are generated separately by the CS and the LURK client.

In the c_hand_and_app_secret, the handshake field contains encrypted messages. These messages are contained in a TLSCiphertext structure, that contains an TLSInnerPlaintext structure. The type of the TLSInnerPlaintext structure MUST be set to 'handshake' otherwise an invalid_handshake error is returned.

psk_proposed, psk_accepted, ...

LURK exchange	TLS handshake	CS operations
s_init_cert_verify	ClientHello ... later of server EncryptedExtensions / CertificateRequest	a,b,c,d
s_init_early_secret	ClientHello	a
s_hand_and_app_secret	ServerHello ... later of server EncryptedExtensions / CertificateRequest	b,
s_new_ticket	earlier of client Certificate / client CertificateVerify / Finished ... Finished	
c_binder_key	-	
c_init_cert_verify	ClientHello...server Finished	e,f
c_init_post_hand_auth	ClientHello ... ServerHello CertificateRequest	e
c_post_hand_auth	CertificateRequest	
c_init_ephemeral	Partial ClientHello	
c_init_early_secret	Partial ClientHello	
c_hand_and_app_secret	ServerHello, {EncryptedExtensions} ... later of { server Finished } / EndOfEarlyData	
c_register_ticket	-	

Figure 4: handshake values per LURK exchange

6.3. session_id

The session_id is a 32 bit identifier that identifies a LURK session between a LURK client and a CS. Unless the exchange is sessionless, the session_id is negotiated at the initiation of the LURK session where the LURK client (resp. the CS) indicates the value to be used for inbound session_id in the following LURK exchanges.
For other LURK exchanges, the session_id is set by the sender to the inbound value provided by the receiving party. When the CS receives an unexpected session_id the CS SHOULD return an invalid_session_id error.

Table Figure 5 indicates the presence of the session_id.

LURK exchange	session_id
s_init_cert_verify	*
s_init_early_secret	y
s_hand_and_app_secret	y
s_new_ticket	y
c_binder_key	-
c_init_early_secret	y
c_init_hand_secret	-
c_hand_secret	y
c_app_secret	y
c_cert_verify	y
c_register_ticket	y
c_post_hand	y

y indicates the session_id is present
- indicates session_id may be absent
* indicates session_id may be present

Figure 5: session_id in LURK exchanges

The session_id structure is defined below: ~~~ uint32 session_id ~~~

6.4. freshness

The freshness function implements perfect forward secrecy (PFS) and prevents replay attack. On the TLS server, the CS generates the ServerHello.random of the TLS handshake that is used latter to derive the secrets. The ServerHello.random value is generated by the CS using the freshness function and the ServerHello.random provided by the LURK client in the handshake structure. The CS operates similarly on the TLS client and generates the ClientHello.random of

the TLS handshake using the freshness function as well as the ClientHello.random value provided by the LURK client in the handshake structure.

If the CS does not support the freshness, the CS SHOULD return an invalid_freshness error. In this document the freshness function is implemented by applying sha256.

Table {table:freshness} details the exchanges that contains the freshness structure.

LURK exchange	freshness
s_init_cert_verify	y
s_init_early_secret	-
s_hand_and_app_secret	y
s_new_ticket	-
c_init_early_secret	y
c_init_hand_secret	y
c_hand_secret	-
c_app_secret	-
c_cert_verify	-
c_register_ticket	-
c_post_hand	-

y indicates freshness is present
- indicates freshness is absent

Figure 6: freshness in LURK exchange

The extension data is defined as follows:

```
enum { sha256(0) ... (255) } Freshness;
```

When the CS is running on the TLS server, the ServerHello.random is generated as follows:

```
server_random = ServerHello.random
ServerHello.random = freshness( server_random + "tls13 pfs srv" );
```

When the CS is running on the TLS client, the ClientHello.random is generated as follows:

```
client_random = ClientHello.random
ClientHello.random = freshness( client_random + "tls13 pfs clt" );
```

The server_random (resp client_random) MUST be deleted once it has been received by the CS.
In some cases, especially when the TLS client enables post handshake authentication and interacts with the CS via a (c_init_post_hand_auth) exchange, there might be some delay between the ClientHello is sent to the server and the Handshake context is shared with the CS. The client_random MUST be kept until the post-handshake authentication is performed as the full handshake is provided during this exchange.

6.5. ephemeral

The Ephemeral structure carries the necessary information to generate the (EC)DHE shared secret used to derive the secrets. This document defines the following ephemeral methods to generate the (EC)DHE shared secret:

- secret_provided: Where (EC)DHE keys and shared secret are generated by the TLS server and provided to the CS
- secret_generated: Where the (EC)DH keys and shared secret are generated by the CS.
- no_secret: where no (EC)DHE is involved, and PSK authentication is performed.

6.5.1. shared_secret_provided:

When ECDHE shared secret are generated by the TLS server, the LURK client provides the shared secret value to the CS. The shared secret is transmitted via the SharedSecret structure, which is similar to the key_exchange parameter of the KeyShareEntry described in [RFC8446] section 4.2.8.
The CS MUST NOT return any data.

```
struct {
   NamedGroup group;
   opaque shared_secret[coordinate_length];
} SharedSecret;
```

Where coordinate_length depends on the chosen group. For secp256r1, secp384r1, secp521r1, x25519, x448, the coordinate_length is respectively 32 bytes, 48 bytes, 66 bytes, 32 bytes and 56 bytes.
Upon receiving the shared_secret, the CS MUST check group is proposed in the KeyShareClientHello and agreed in the KeyShareServerHello.

6.5.2. secret_generated:

When the ECDHE public/private keys are generated by the CS, the LURK client requests the CS the associated public value. Note that in such cases the CS would receive an incomplete Handshake Context from the LURK client with the public part of the ECDHE missing. Typically the ServerHello message would present a KeyShareServerHello that consists of a KeyShareEntry with an empty key_exchange field, but the field group is present.

The CS MUST check the group field in the KeyShareServerHello, and get the public value of the TLS client from the KeyShareClientHello. The CS performs the same checks as described in [RFC8446] section 4.2.8. The CS generates the private and public (EC)DH keys, computes the shared key and return the KeyShareEntry server_share structure defined in [RFC8446] section section 4.2.8 to the LURK client.

6.5.3. no_secret

With PSK authentication, (EC)DHE keys and shared secrets are not needed. The CS SHOULD check the PSK authentication has been agreed, that is pre_shared_key and psk_key_exchamge_modes extensions are noth present in the ClientHello and in the ServerHello

When the ephemeral method or the group is not supported, the CS SHOULD return an invalid_ephemeral error.

LURK exchange	ephemeral
s_init_early_secret	-
s_init_cert_verify	y+
s_hand_and_app_secret	y
s_new_ticket	-
c_init_early_secret	no secret_provided	c_init_ephemeral no secret_generated
c_init_hand_secret	y+
c_hand_secret	y
c_app_secret	-
c_cert_verify	-
c_register_ticket	-
c_post_hand	-

y indicates ephemeral is present
y+ indicates ephemeral is present with ephemeral_method different from no_secret.
- indicates ephemeral is absent

Figure 7: Ephemeral field in LURK exchange

The extension data is defined as follows:

```
enum { no_secret (0), secret_provided(1), secret_generated(2) (255)}
EphemeralMethod;

EphemeralRequest {
    EphemeralMethod method;
    select(method) {
        case secret_provided:
            SharedSecret shared_secret<0..2^16>;
    }
}

EphemeralResponse {
    select(method) {
        case secret_generated:
            KeyShareEntry server_share
  }
}
```

6.6. selected_identity

The selected_identity indicates the identity of the PSK used in the key schedule. The selected_identity is expressed as a (0-based) index into the identities in the client's list. The client's list is provided in the pre_shared_key extension as expressed in [RFC8446] section 4.2.11.

The LURK client MUST provide the selected_identity only when PSK or PSK-authentication is envisioned and when the PSK has not been provided earlier. These exchanges are s_init_early_secret on the TLS server and c_init_early_secret and c_init_hand_secret on the TLS client side.

LURK exchange	req	resp
s_init_early_secret	M	-
s_init_cert_verify	-	-
s_hand_and_app_secret	-	-
s_new_ticket	-	-
c_init_early_secret	-	-
c_init_hand_secret	-	-
c_hand_secret	-	-
c_app_secret	-	-
c_cert_verify	-	-
c_register_ticket	-	-
c_post_hand	-	-

M indicates the field is mandatory
M* indicates the field is mandatory but psk may be void
- indicates the field MUST NOT be provided

Figure 8: psk_id in LURK exchange

The extension data is defined as follows:

uint16 selected_identity; //RFC8446 section 4.2.11

The CS retrieve the PSK identity from the ClientHello and SHOULD send an invalid_psk error if an error occurs. If the PSK is not provided, a default PSK is generated as described in [RFC8446] section 7.1. If the default PSK is not allowed then an invalid_psk is returned.

6.7. certificate

The certificate field is used by the LURK client to indicate the presence with associated value or absence of certificate in the TLS exchange. When necessary, the CS is expected to generate the appropriated message for the Handshake Context.

Upon receiving a certificate field, the CS MUST: 1. ensure the presence or absence of certificate is coherent with the handshake messages (see Section 6.7.1). 2. when the certificate is provided the CS checks the value corresponds to an acceptable pre-provisionned value (see Section 6.7.2). 3. when the certificate is provided, the CS MUST generate the appropriated corresponding message (see Section 6.7.3).

If the CS is not able to understand the lurk_tls13_certificate field, it SHOULD return an invalid_certificate error.

Table Figure 9 indicates the presence of that field in the LURK exchanges. The

LURK exchange	req	resp	certificate type
s_init_early_secret	-	-	
s_init_cert_verify	M	-	server certificate
s_hand_and_app_secret	-	-	
s_new_ticket	M*	-	client certificate
c_init_early_secret	-	-	
c_init_hand_secret	-	-	
c_hand_secret	-	-	
c_app_secret	M	-	server certificate
c_cert_verify	-	-	
c_register_ticket	M*	-	client certificate
c_post_hand	-	-	

(*) indicates the field MAY be emtpy.
M indicates the field is mandatory
- indicates the field MUST NOT be provided

Figure 9: tag per LURK exchange

There are different ways the LURK client can provide the certificate message:

```
enum { empty(0), finger_print(1), uncompressed(2), (255)
}; LURKTLS13CertificateType

struct {
    LURKTLS13CertificateType certificate_type;
    select (certificate_type) {
        case empty:
          // no payload
        case finger_print
            uint32 hash_cert;
        case uncompressed:
            Certificate certificate; // RFC8446 section 4.4.2
    };
} LURKTLS13Certificate;
```

empty indicates there is no certificates provided by this field.

fingerprint a 4 bytes finger print length that represents the fingerprinting of the TLS Certificate message. Fingerprinting is described in [RFC7924] and takes as input the full handshake

message - that is a message of message type certificate with that contain the Certificate as its message_data. In this document only the 4 most left bytes of the output are considered. uncompressed

indicates the Certificate message as defined in [RFC8446] is provided. compressed

indicates the CompressedCertificate [I-D.ietf-tls-certificate-compression]

6.7.1. presence or absence of certificate message

The absence of the certificate is indicated by a lurk_certificate_type set to 'empty'. The presence is indicated by a lurk_certificate_type set to 'uncompressed', 'compressed' or 'finger_print'. The absence of a server Certificate message is only acceptable when PSK or PSK-ECDHE authentication are used which is indicated by the presence of a psk_key_exchange_modes and a pre_shared_key extension in the ServerHello and ClientHello message. If the lurk_certificate_type is set to empty and these extensions are not found an invalid_certificate error SHOULD be raised. If the lurk_certificate_type is not set to empty and these extensions are found a invalid_certificate error SHOULD be raised.

The presence of a client Certificate message is only acceptable when a CertificateRequest message is found in the ServerHello message. If the lurk_certificate_type is set to empty and a CertificateRequest is present in the ServerHello an invalid_certificate error SHOULD be raised. If the lurk_certificate_type is not set to empty and a CertificateRequest is not present in teh ServerHello message, an invalid_certificate error SHOULD be raised.

6.7.2. certificate field validation

The certificate field can be used for client certificate or for server certificate. That certificate is called the designated certificate.
The LURK Client indicates the certificate material either by providing the uncompressed certificate or via a finger_print.

The LURK client MAY provide a certificate field of type uncompressed to either carry the Certificate data of the designated certificate or the necessary data to derive the Certificate data. Typically, providing Certificate as defined in [RFC8446] will enable to CS to generate Certificate messages as defined in [RFC8446].

The LURK client MAY provide a certificate field of type finger_print in conjunction of additional information shared between the LURK client and the CS. The finger_print is a 4 byte hint derived from the 4 most significant bytes of the fingerprint as defined in [RFC7924].

As a result, the certificate field will be validated if one of the following condition applies: 1. the certificate_type is set to 'uncompressed' or 'compressed' and the stored certificate is as described in [RFC8446]. 2. the certificate_type is set to 'finger_print' and the designated certificate has been provisioned.

6.7.3. generation of the certificate message

The TLS exchange carries the certificate information either in a unaltered Certificate message [RFC8446], or in a CompressedCertificate message [I-D.ietf-tls-certificate-compression].

The CompressedCertificate message is decompressed and the uncompressed Certificate message is considered ni the TLS handshake. As a result, on a CS point of view the use of a CompressedCertificate message does not impact the handshake transcript.

6.8. tag

This field provides extra information. Currently, this fields is used by the LURK client or the CS to indicate the session is ended. Table Figure 10 indicates the tag values and Table Figure 11 the LURK messages that contains the tag field.

When the LURK client knows this will be the last LURK exchange performed within a given session, the LURK client sets the last_exchange bit. When the CS receives a last_exchange set, the CS answers normally but clear the session right after the response has been sent. Similarly, when the CS knows no further LURK exchanges will be accepted within a session, the CS sets the last_exchange bit in the response. Upon receiving the response, the LURK client does not proceed to additional LURK exchange.

Bit	description
0	last_exchange
1-7	RESERVED

Figure 10: tag description

LURK exchange	req	resp
s_init_early_secret	-	-
s_init_cert_verify	M	M
s_hand_and_app_secret	M	M
s_new_ticket	M	M
c_init_early_secret	M	M
c_init_hand_secret	M	M
c_hand_secret	M	M
c_app_secret	M	M
c_cert_verify	M	M
c_register_ticket	M	M
c_post_hand	M	M

M indicates the field is mandatory
- indicates the field MUST NOT be provided

Figure 11: tag per LURK exchange

6.9. secret

The Secret structure is used by the CS to send the various secrets derived by the key schedule described in [RFC8446] section 7.

```
enum {
    binder_key (0),
    client_early_traffic_secret(1),
    early_exporter_master_secret(2),
    client_handshake_traffic_secret(3),
    server_handshake_traffic_secret(4),
    client_application_traffic_secret_0(5),
    server_application_traffic_secret_0(6),
    exporter_master_secret(7),
    esumption_master_secret(8),
    (255)
} SecretType;

struct {
    SecretType secret_type;
    opaque secret_data<0..2^8-1>;
} Secret;
```

secret_type: The type of the secret or key

secret_data: The value of the secret.

6.10. signature

The signature requires the signature scheme, a private key and the appropriated context. The signature scheme is provided using the SignatureScheme structure defined in [RFC8446] section 4.2.3, the private key is derived from the lurk_tls13_certificate Section 6.7 and the context is derived from the handshake Section 6.2 and lurk_tls13_certificate Section 6.7.

Signing operations are described in [RFC8446] section 4.4.3. The context string is derived from the role and the type of the LURK exchange as described below. The Handshake Context is taken from the key schedule context.

type	context
s_init_cert_verify	"TLS 1.3, server CertificateVerify"
c_cert_verify	"TLS 1.3, client CertificateVerify"

```
struct {
    opaque signature<0..2^16-1>; //RFC8446 section 4.4.3.
} Signature;
```

7. LURK exchange on the TLS server

This section describes the LURK exchanges that are performed on the TLS server. Unless specified used structures are described in Section 6 The state diagram is provided in section Section 12.2

7.1. s_init_early_secret

s_init_early_secret initiates a LURK session when the server is authenticated by the PSK or PSK-ECDHE methods. This means the ClientHello received by the TLS server and ServerHello responded by the TLS server MUST carry the pre_shared_key and psk_key_exchange_modes extensions.

selected_identity indicates the selected PSK

```
struct{
    uint32 session_id
    FreshnessFunct freshness
    uint16 selected_identity
    Handshake handshake<0..2^32> //RFC8446 section 4
    uint16 secret_request;
}SInitEarlySecretRequest

struct{
    uint32 session_id
    Secret secret_list<0..2^16-1>;
}SInitEarlySecretResponse
```

The binder_key MUST be requested, since it is used to validate the PSK. The TLS client MAY indicate support for early application data via the early_data extension. Depending on the TLS server policies, it MAY accept early data and request the client_early_traffic_secret. The TLS server MAY have specific policies and request early_exporter_master_secret.

The CS MUST check pre_shared_key and psk_key_exchange_modes extensions are present in the ClientHello message. If these extensions are not present, a invalid_handshake error SHOULD be returned. The CS MUST ignore the client_early_traffic_secret if early_data extension is not found in the ClientHello. The Cryptographic Service MAY ignore the request for client_early_traffic_secret or early_exporter_master_secret depending on configuration parameters.

7.2. s_init_cert_verify

s_init_cert_verify initiates a LURK session when the server is authenticated with ECDHE. The ClientHello received by the TLS server, and the ServerHello and optionally the HelloRetryRequest MUST carry a key_share extension.

If the LURK client is configured to not proceed to further exchange, it sets the last_exchange bit of the tag. When this bit is set, the session_id is ignored. The CS sets the last_exchange bit if the last_Exchange bit has been set by the LURK client or when it has been configured to not accept further LURK exchange.

```
struct{
    uint8 tag;
    select tag.last_exchange){
      case False:
        uint32 session_id;
    }
    FreshnessFunct freshness;
    Ephemeral ephemeral;
    Handshake handshake<0..2^32>; //RFC8446 section 4
    LURKTLS13Certificate certificate;
    uint16 secret_request;
    SignatureScheme sig_algo; //RFC8446 section 4.2.3.
}SInitCertVerifyRequest

struct{
    uint8 tag;
    select tag.last_exchange){
      case False:
        uint32 session_id;
    }
    Ephemeral ephemeral;
    Secret secret_list<0..2^16-1>;
    Signature signature;
}SInitCertVerifyResponse
```

sig_algo SignatureScheme is defined in [RFC8446] section 4.2.3.

7.3. s_hand_and_app_secret

The s_hand_and_app_secret is necessary to complete the ServerHello and always follows an s_init_early_secret LURK exchange. Such sequence is guaranteed by the session_id. In case of unknown session_id or an invalid_request error SHOULD be returned.

The LURK client MUST ensure that PSK or PSK-ECDHE authentication has been selected via the presence of the pre_shared_key extension in the ServerHello. In addition, the selected identity MUST be the one provided in the pre_shared_key extension of the previous s_init_early_secret exchange. The CS MUST also check the selected cipher in the ServerHello match the one associated to the PSK. The CS generates the Finished message as described in [RFC8446] section 4.4.4. Which involves the h_s secret. The LURK client MAY request the exporter_master_secret depending on its policies. The CS MAY ignore the request based on its policies.

If the LURK client is configured to not proceed to further exchange, it sets the last_exchange bit of the tag. The CS sets the last_exchange bit if the last_exchange bit has been set by the LURK

client or when it has been configured to not accept further LURK exchange.

```
struct{
    uint8 tag
    uint32 session_id
    Ephemeral ephemeral
    Handshake handshake<0..2^32> //RFC8446 section 4
    uint16 secret_request;
} SHandAndAppSecretRequest

struct{
    uint8 tag
    uint32 session_id
    Ephemeral ephemeral
    Secret secret_list<0..2^16-1>;
} SHandAndAppSecretResponse
```

7.4. s_new_tickets

new_session ticket handles session resumption. It enables to retrieve NewSessionTickets that will be forwarded to the TLS client by the TLS server to be used later when session resumption is used. It also provides the ability to delegate the session resumption authentication from the CS to the TLS server. In fact, if the LURK client requests and receives the resumption_master_secret it is able to emit on its own NewSessionTicket. As a result s_new_ticket LURK exchanges are only initiated if the TLS server expects to perform session resumption and the CS responds only if if session_resumption is enabled.

The CS MAY responds with a resumption_master_secret based on its policies.

The LURK client MAY perform multiple s_new_ticket exchanges. The LURK client and CS are expected to advertise by setting the last_exchange bit in the tag field.

```
struct {
    uint8 tag
    uint32 session_id
    Handshake handshake<0..2^32> //RFC8446 section 4.
    LURKTLS13Certificate certificate;
    uint8 ticket_nbr;
    uint16 secret_request;
} SNewTicketRequest;


struct {
    uint8 tag
    uint32 session_id
    Secret secret_list<0..2^16-1>;
    NewSessionTicket ticket_list<0..2^16-1>; //RFC8446 section 4.6.1.
} SNewTicketResponse;
```

ticket_nbr: designates the requested number of NewSessionTicket. In the case of delegation this number MAY be set to zero. The CS MAY responds with less tickets when the value is too high.

8. LURK exchange on the TLS client

This section describes the LURK exchanges that are performed on the TLS server. The state diagram is provided in section Section 12.1

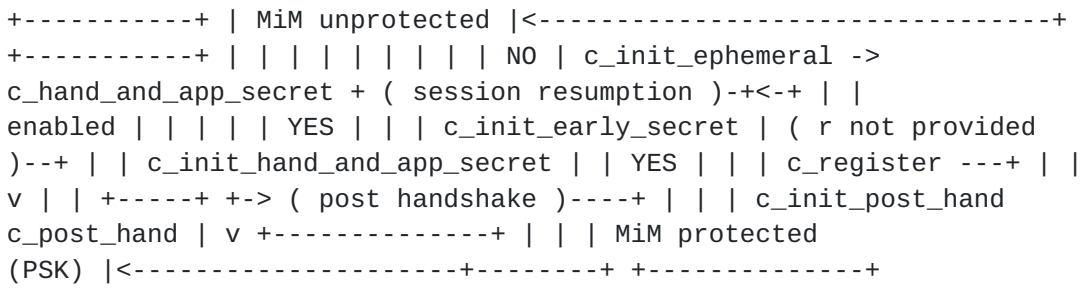
The LURK exchanges on the TLS client are determined by the following parameters 1. the ability of the TLS client to authenticate with a CertificateVerify 2. the ability of the TLS client to perform post handshake authentication (indicated by a post_handshake_auth extension in the ClientHello) 3. the authentication methods chosen by the server (PSK, PSK-ECDHE, ECDHE) 4. the owner of the private key of the shared secret (TLS client, CS) 5. the owner of the session resumption secret (TLS client or CS)

When the TLS client does not provide the ability to authenticate itself (no CertificateVerify nor post handshake authentication) , the CS does not hold the credentials of the TLS client (a PSK or a private key). In this case, the only method to authenticate the server is ECDHE. The owner of the private can be CS or the TLS client. When the owner of the private key is the TLS client, it will be able to generate the shared secret (with its private key) and derive all necessary secrets . As a result, no interaction is needed between the TLS client and the CS. The TLS client owns the session resumption secret and so further credential will not be protected, and thus fall outside the scope of this document. When the owner of the private key is the CS, the TLS client will need to retrieve the public key via c_init_ephemeral LURK exchange to complete its

ClientHello. Then additional secrets are necessarily computed by the CS - as the keyschedule takes the shared secret as input - and retrieved by the TLS client via the c_hand_and_app_secret LURK exchange. If the CS allows the retrieval by the TLS client of the session resumption secret the credentials used for the next sessions will not be protected by the CS and thus fall outside the scope of this document. On the other hand, if the CS does not allow the retrieval of the session secret, then session resumption requires the TLS client to provide the NewSessionTicket provided by the TLS server to the CS via the c_register LURK exchange and the credentials (PSK) are protected by the CS for the next sessions.

When the TLS the TLS client provides the ability to authenticate itself. This can be achieved via a certificate or through a PSK. Note that the two authentications can be done together for example a post handshake authentication combined with a PSK or PSK-ECDHE.
We assume the private key associated to the client certificate is protected by the CS. The ability to perform a post handshake authentication (c_init_post_hand or c_post_hnad) ensures that the owner of the private key is aware of the handshake.
Note that it does not provides protection of PFS via the guarantee of ephemeral secrets - instead PFS is provided by preventing the exchange to be replayed.

NOTE: REDO the ANALYSE. SECURE/UNSECURE MAY NOT BE SUFFICIENT AS THERE ARE: * protection of the credentials (PSK, private key) -> PSK is in the CS if 'r' secret is not provided. * protection associated to the session ** GUARANTEE OF EPHEMERAL SECRETS: makes sure the ephemeral secrets are deleted. Recording the ecdhe private information enables the key schedule to be performed for ECDHE. Adds additional randomness for PSK-ECDHE ** AWARENESS OF THE HANDSHAKE. Someone else may also use that exchange... but I do not see exactly how. ** ...



c_init_hand_and_app_secret (eph=provided) -> (c_post_hand. c_register_ticket) c_init_ephemeral (eph=generated)-> c_hand_and_app_secret -> (c_post_hand. c_register_ticket) c_init_early_secret (prov, gen) ^

c_init_post_hand : unable to bind the handshake messages to the key used to generates the hash. SHOUDL we consider this ?

8.1. c_init_cert_verify

The c_init_hand_and_app_secret LURK exchange used under the three following conditions: TLS client authenticates the TLS server using ECDHE, the TLS client has generated the ECDHE private key - as opposed to the CS -, and the TLS server requires the TLS client to authenticate with a CertificateVerify during the TLS key exchange.

The last_message is set as defined in Section 6.8 but with the additional condition that a post handshake authentication may be performed. More precisely, the TLS client sets the last_exchange only if a post handshake authentication may be performed in the future. The CS sets the last_message only if post handshake is enabled by the CS and the post_handshake_authentication extension is present in the ClientHEllo.

The freshness is handled as described in Section 6.4.

The Ephemeral MUST be set to 'secret_provided' as the TLS client has generated the ECDHE private key. The CS treats the ephemeral as describe din Section 6.5.

The handshake MUST NOT contain any of the pre_shared_key and psk_key_exchange_modes extensions in the ServerHello. If these extensions are found, an invalid_handshake error is returned. The handshake MUST contain the key_share extensions in the ClientHello and the ServerHello and returns an invalid handshake error otherwise. The handshake MUST contain a CertificateRequest and returns a invalid_handshake error otherwise.

No secret are generated. The TLS client has all the necessary material for the key schedule and as such can proceed to session resumption.

This exchange is followed by a c_post_hand_auth exchange.

When the exchange is not terminal, i.e. the las_exchange is unset, the CS generates the client Finished message.

```
struct{
    uint8 tag;
    select tag.last_exchange){
      case False:
        uint32 session_id;
    }
    Freshness freshness
    Ephemeral = provided
    Handshake handshake<0..2^32> //RFC8446 section 4 (clear)
    SignatureScheme sig_algo;
    LURKTLS13Certificate cert;
}CInitCertVerifypRequest


struct{
    uint8 tag;
    select tag.last_exchange){
      case False:
        uint32 session_id;
    }
    FreshnessFunct freshness;
    Signature signature
}CInitCertVerifyResponse
```

8.2. c_init_post_hand_auth

The c_init_post_hand_auth happens when the TLS client is authenticating using a post handshake authentication and all previous key exchanges messages with the TLS server did not result in the creation of a session. As mentioned in Section 6.4 the creation of an earlier session with the CS will end up in the TLS client not knowing the value of the client_random, making this exchange impossible. As a result, this exchange is expected under the following conditions. The TLS client is authenticating the TLS server via ECDHE (or PSK /PSK-ECDHE with an unprotected PSK), the TLS client has generated the ephemerals private key and derived all secrets. As the TLS client may need to perform multiple authentications, the c_init_post_hand_auth exchange may be followed by additional c_post_hand_auth.

Upon receiving the request, the CS checks the presence of the post_handshake_auth extension in the ClientHello. The CS also checks the presence of a CertificateRequest message after the client Finished message. If the extension or message is not found, and invalid_handshake error is returned.

The ephemeral mode MUST be secret_provided or no_secret. If other methods are found, an invalid_ephemeral is returned.

```
struct{
    Tag tag
    Freshness freshness
    Ephemeral = provided
    Handshake handshake<0..2^32> //RFC8446 section 4 (clear)
clientHello...client finished CertificateRequest
    SignatureScheme sig_algo;
    LURKTLS13Certificate cert;
}CInitPostHandAuthRequest


struct{
    Tag tag
    Signature signature
}CInitPostHandAuth
```

8.3. c_post_hand_auth

The c_post_hand_auth exchange enables a TLS client to perform post handshake authentication. It follows a c_init_post_hand_auth, c_init_cert_verify a c_hand_and_app or c_register_ticket.

```
struct{
    Tag tag
    Handshake handshake<0..2^32> // CertificateRequest
    SignatureScheme sig_algo;
    LURKTLS13Certificate cert;
}CInitPostHandAuthRequest


struct{
    Tag tag
    Signature signature
}CInitPostHandAuth
```

8.4. c_init_ephemeral

The c_init_ephemeral LURK exchange is performed when the TLS client authenticates the TLS server using ECDHE with an ephemeral value generated by the CS.

The ephemeral method MUST be set to 'secret_generated' otherwise an invalid_ephemeral error is returned. The Handshake value is a partial ClientHello with a key_share extension that doe not contain a ECHDE public value. The Client Hello MUST not have pre_shared_key or psk_key_exchange_mode. If any of these condition is not met a invalid_handshake error is returned.

```
struct{
    uint32 session_id
    Freshness freshness
    Ephemeral ephemeral
    Handshake handshake<0..2^32> //RFC8446 section 4
}CInitEphemeral

struct{
    uint32 session_id
    Ephemeral ephemeral
}CInitEphemeral
```

8.5. c_init_early_secret

The c_init_early_secret LURK exchange initiates a LURK session when the TLS client proposes at least one of PSK for PSK or PSK-ECDHE authentication method.

This exchange differs to the s_init_early_secret in two aspects: First, a partial ClientHello (as described in [RFC8446] section 4.2.11.2) with a potentially partial key_share extension (as described in Section 6.5. As a result, the CS needs to complete it to derive the full ClientHello. Second, the CS does not generate the secrets for a single chosen psk identity (selected_identity), but for all PSK identities specified in the PreSharedKeyExtension.identities.

The shared secret of the exchange can be absent when PSK only is proposed in which case the ephemeral method is set to 'no_secret'. When PSK-ECDHE or ECDHE is proposed, the private key associated to the shared secret can be derived by the CS or the TLS client. When derived by the CS, the ephemeral_method is set to 'secret_generated'. When generated by the TLS client, the ephemeral method is set to 'no_secret' as teh shared secret cannot be provided yet and will be provided once the ServerHello has been received by the TLS client.

The handshake is a partial ClientHello as described in [RFC8446] section 4.2.11.2. The CS checks PSK or PSK-ECDHE is proposed and returns an invalid_handshake error otherwise. The CS computes the secrets associated to each identity. The secret_list_list contains the list of secret_lists associated to each identity in the same order as these identities. If an identity is unknown an invalid_handshake error is returned. The CS SHOULD limit the number of identities and if that number is exceeded, an too_many_identities SHOULD be returned. Secrets are generated as described in Section 6.1.

```
struct{
    uint32 session_id
    Freshness freshness
    Ephemeral ephemeral
    Handshake handshake<0..2^32> //RFC8446 section 4
    uint16 secret_request;
}CInitEarlySecretRequest

SecretList secret_list<0..2^16-1>;

struct{
    uint32 session_id
    Ephemeral ephemeral
    SecretList secret_list_list<0..2^16-1>;
}CInitEarlySecretResponse
```

8.6. c_hand_and_app_secret

The c_hand_and_app LURK exchange always follows a c_init_early_secret or a c_init_ephemeral LURK exchange.

The tag field may have the last_exchange bit set to indicate that no further exchange is expected. No further exchange means that the TLS client does not intend to perform session resumption nor to perform post handshake authentication. A TLS client SHOULD NOT set the last_exchange bit if a post_handshake_auth extension is present in its ClientHello.

If the LURK exchange follows a c_init_ephemeral, the ECDHE private key was generated by the CS during the c_init_ephemeral_exchange. The CS checks the ephemeral_method is set to 'no_secret'.

If the LURK exchange follows a c_init_early_secret, the ephemeral_method had been set to 'no_secret' or 'secret_generated'. If the TLS server is authenticated using PSK or if the ephemeral_method was previously set to 'secret_generated', the CS checks the ephemeral_method is set to 'no_secret'. If the TLS server is authenticated using PSK-ECDHE or ECDHE or if the ephemeral_method was previously set to 'no_secret', the CS checks the ephemeral_method is set to 'secret_provided'. If a mismatch is found between the ephemeral_method and the selected authentication method, an invalid_ephemeral

The handshake field contains the ServerHello and other encrypted messages. Upon receiving a request the CS determines which secrets needs to be generated as described in Section 6.1. The generation of these secret requires the shared secret to be generated - including the default value for the PSK authentication. The derivation of the

handshake secrets (h_s, h_c) do not need to decrypt the encrypted messages.

However, the derivation of the application secrets (a_s, a_c), export secret (x) or resumption secret (r) do. If these secrets are requested, the CS needs to generate the handshake secrets, the server_handshake_traffic_secret as described in [RFC8446] section 7.3 to decrypt the encrypted messages. The CS can then generate all secrets except the resumption secret.

If session resumption or post handshake is not explicitly prohibited, by setting the last_exchange of the tag field, the CS generates all missing messages until the client Finished. Otherwise, the client Finished message MAY not be generated.

If a CertificateRequest is present, the Certificate and CertificateVerify needs to be generated. Unlike on the TLS server, where the TLS server indicates the certificate to chose as well as the signature scheme to select, on the TLS client, such decision is left to the CS. The choice of the signature algorithm and certificate is performed by the CS as described in [RFC8446] section 4.4.2.3.
The Certificate, respectively CertificateVerify and Finished message are generated as described in [RFC8446] section 4.4.2, section 4.4.3, and section 4.4.4.

```
struct{
    uint8 tag
    uint32 session_id
    Ephemeral ephemeral
    Handshake handshake<0..2^32> //RFC8446 section 4
    uint16 secret_request;
}CHandAndAppRequest

struct{
    uint8 tag
    uint32 session_id
    LURKTLS13Certificate certificate;
    SignatureScheme sig_algo
    Signature signature
    Secret secret_list<0..2^16-1>;
}CHandAndAppRequest
```

8.7. c_register_tickets

The c_register_ticket is only used when the TLS client intend to perform session resumption. The LURK client MAY provide one or multiple NewSessionTickets. These tickets will be helpful for the session resumption to bind the PSK value to some identities.

```
struct {
    uint8 tag
    uint32 session_id
    NewSessionTicket ticket_list<0..2^16-1>; //RFC8446 section 4.6.1.
} RegisterTicketRequest;

struct {
    uint8 tag
    uint32 session_id
} RegisterTicketResponse;
```

9. Security Considerations

Security credentials as per say are the private key used to sign the CertificateVerify when ECDHE authentication is performed as well as the PSK when PSK or PSK-ECDHE authentication is used.

The protection of these credentials means that someone gaining access to the CS MUST NOT be able to use that access from anything else than the authentication of an TLS being established. In other way, it MUST NOT leverage this for: * any operations outside the scope of TLS session establishment. * any operations on past established TLS sessions * any operations on future TLS sessions * any operations on establishing TLS sessions by another LURK client.

The CS outputs are limited to secrets as well as NewSessionTickets. The design of TLS 1.3 make these output of limited use outside the scope of TLS 1.3. Signature are signing data specific to TLS 1.3 that makes the signature facility of limited interest outside the scope of TLS 1.3. NewSessionTicket are only useful in a context of TLS 1.3 authentication.

ECDHE and PSK-ECDHE provides perfect forward secrecy which prevents past session to be decrypted as long as the secret keys that generated teh ECDHE share secret are deleted after every TLS handshake. PSK authentication does not provide perfect forward secrecy and authentication relies on the PSK remaining sercet. The Cryptographic Service does not reveal the PSK and instead limits its disclosure to secrets that are generated from the PSK and hard to be reversed.

Future session may be impacted if an attacker is able to authenticate a future session based on what it learns from a current session. ECDHE authentication relies on cryptographic signature and an ongoing TLS handshake. The robustness of the signature depends on the signature scheme and the unpredictability of the TLS Handshake. PSK authentication relies on not revealing the PSK. The CS does not reveal the PSK. TLS 1.3 has been designed so secrets generated do not disclose the PSK as a result, secrets provided by the Cryptographic do not reveal the PSK. NewSessionTicket reveals the identity (ticket) of a PSK. NewSessionTickets.ticket are expected to be public data. It value is bound to the knowledge of the PSK. The Cryptographic does not output any material that could help generate a PSK - the PSK itself or the resumption_master_secret. In addition, the Cryptographic only generates NewSessionTickets for the LURK client that initiates the key schedule with CS with a specific way to generate ctx_id. This prevents the leak of NewSessionTickets to an attacker gaining access to a given CS.

If an the attacker get the NewSessionTicket, as well as access to the CS of the TLS client it will be possible to proceed to the establishment of a TLS session based on the PSK. In this case, the CS cannot make the distinction between the legitimate TLS client and teh attacker. This corresponds to the case where the TLS client is corrupted.

Note that when access to the CS on the TLS server side, a similar attack may be performed. However the limitation to a single re-use of the NewSessionTicket prevents the TLS server to proceed to the authentication.

Attacks related to other TLS sessions are hard by design of TLS 1.3 that ensure a close binding between the TLS Handshake and the generated secrets. In addition communications between the LURK client and the CS cannot be derived from an observed TLS handshake (freshness function). This makes attacks on other TLS sessions unlikely.

10. IANA Considerations

11. Acknowledgments

12. Annex

12.1. LURK state diagrams on TLS client

The state diagram sums up the LURK exchanges. The notations used are defined below:

LURK exchange indicates a LURK exchange is stated by the LURK client or is received by the CS ---> (resp. <---) indicates a TLS message is received (resp. received). These indication are informative to illustrates the TLS state machine.

CAPITAL LETTER indicates potential configuration parameters or policy applied by the LURK client or the CS. The following have been considered:

- PSK, PSK-ECDHE, ECDHE that designates the authentication method. This choice is made by the LURK client. The choice is expressed by a specific LURK exchange as well as from the TLS Handshake Context.

- SESSION_RESUMPTION indicates the session resumption has been enabled on the LURK client or the CS. As a consequence the TLS client is considered performing session resumption and the TLS server MUST make session resumption possible.

- POST_HANDSHAKE_AUTH indicates that post handshake authentication proposed by the TLS client in a post_handshake_auth extension is not ignored by the LURK client or on the CS.

Note that SESSION_RESUMPTION, POST_HANDSAHKE_AUTH are mostly informative and the current specification does not mandate to have such configuration parameters. By default, these SHOULD be enabled.

Other potential configuration could be proposed for configuring LURK client or CS policies. These have not been represented in the state diagram and the specification does not mandate to have these parameters implemented.

- CLIENT_EARLY_TRAFFIC indicates that client early traffic MAY be sent by the TLS client and the notification by the TLS client in the ClientHello via the early_data extension MUST be considered.

- EARLY_EXPORTER_MASTER_SECRET indicates whether or not early_exporter_master_secret MUST be requested by the LURK client and responded by the CS.

- MASTER_EXPORTER indicates whether or not exporter_master_secret MUST be requested by the LURK client and responded by the CS.

- SESSION_RESUMPTION_DELEGATION indicates whether or not session_resumption_master is requested by the LURK client and responded by the CS.

- MAX_SESSION_TICKET_NBR indicates the maximum number of tickers that can be requested or provided by the LURK client and provided by the CS. It is strongly RECOMMENDED to have such limitations being configurable.

The analysis of the TLS Handshake Context enables to set some variables that can be used by the LURK client to determine which LURK exchange to proceed as well as by the CS to determine which secret MAY be responded. The following variables used are:

psk_proposed: The TLS Client is proposing PSK authentication by including a pre_shared_key and a psk_key_exchange_mode extensions in the ClientHello.

dhe_proposed: The received or to be formed ClientHello contains a key_share extensions.

psk_accepted: The chosen authentication method is pSK or PSK-ECDHE which is indicated via the pre_shared_key extension in the ServerHello.

0rtt_proposed: Indicates the TLS client supports early data which is indicated by the early_data extension in the ClientHello.

post_handshake_proposed: indicates the TLS client supports post handshake authentication which is indicated by the presence of a post_handshake_auth extension in the ClientHello.

finished: indicates that the LURK client or the CS has determined the session shoudl be closed an ks_ctx are deleted.

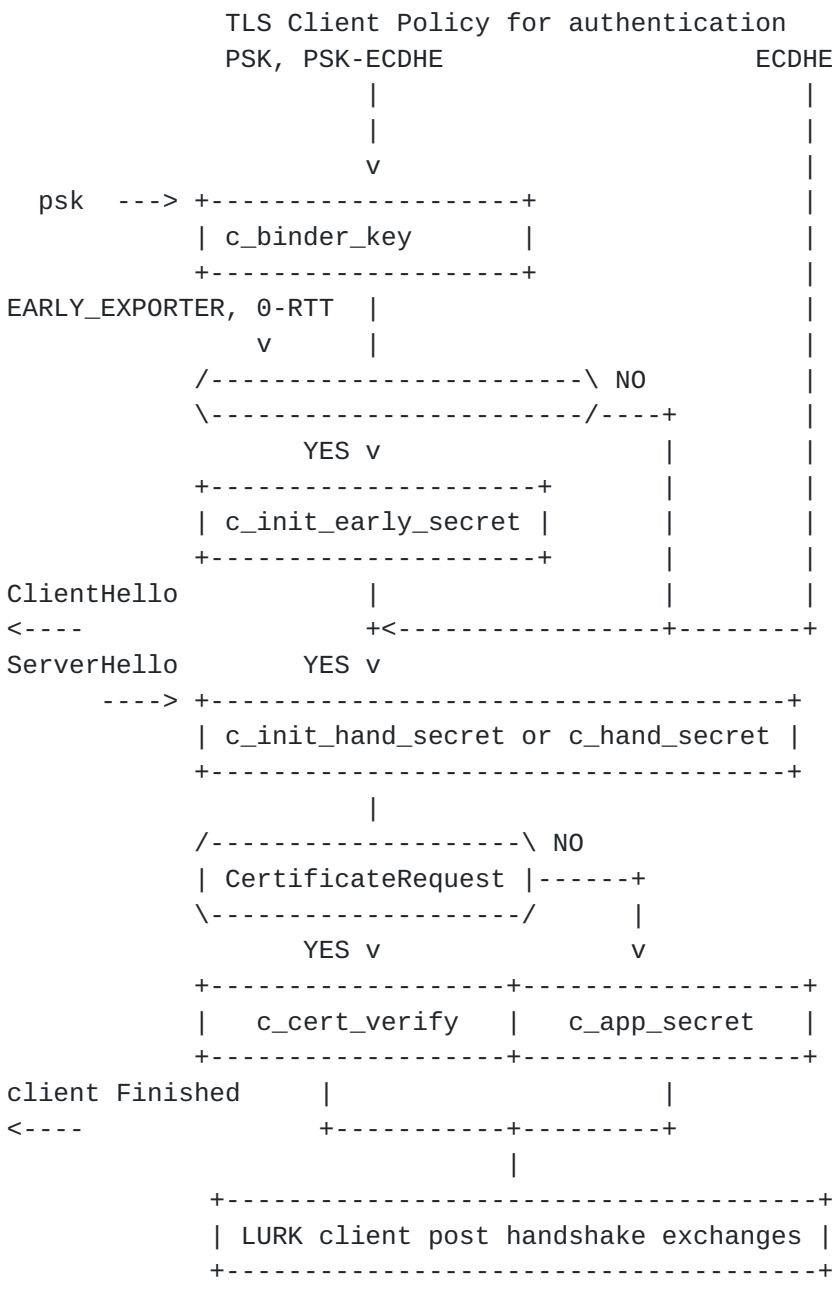
The CS contains three databases:

CTX_ID_DB: database that contains the valid ctx_id of type opaque.

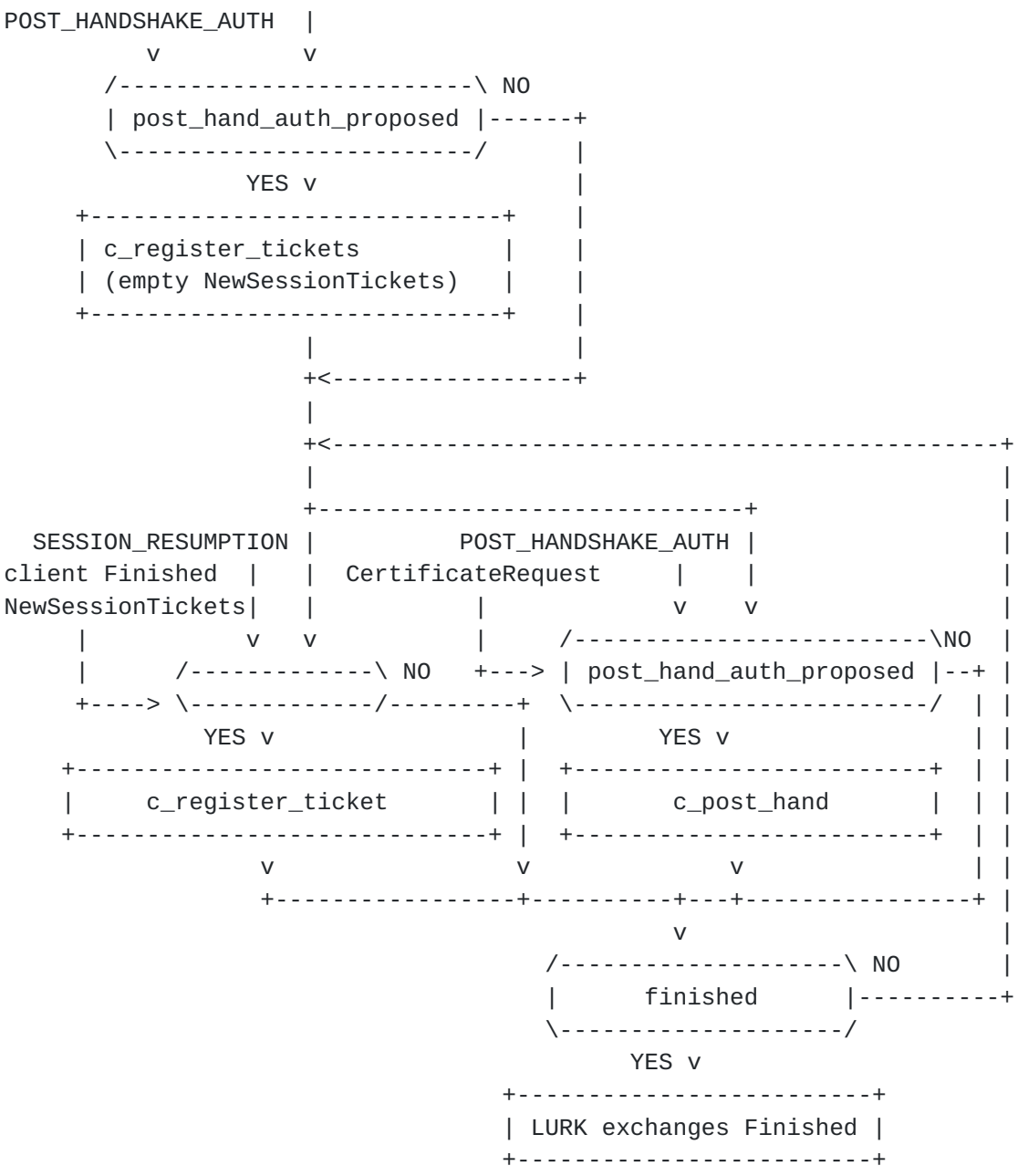
PSK_DB: contains the list of PSKs, with associated parameters such as Hash function. This database includes the session resumption tickets.

Key_DB: contains the asymetric signing keys with supported signing algorithms.

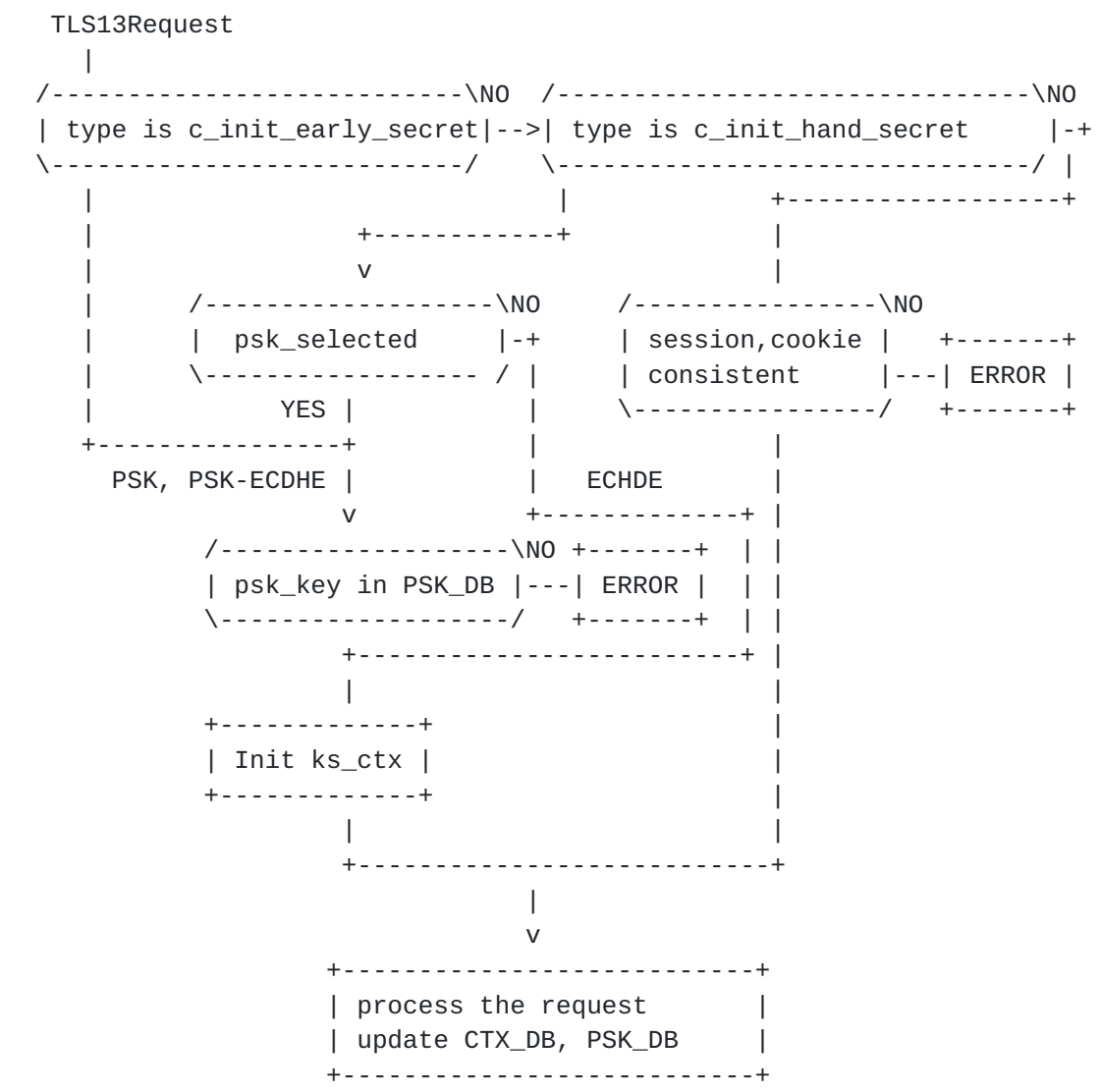
12.1.1. LURK client



The LURK client post handshake diagram is represented below:

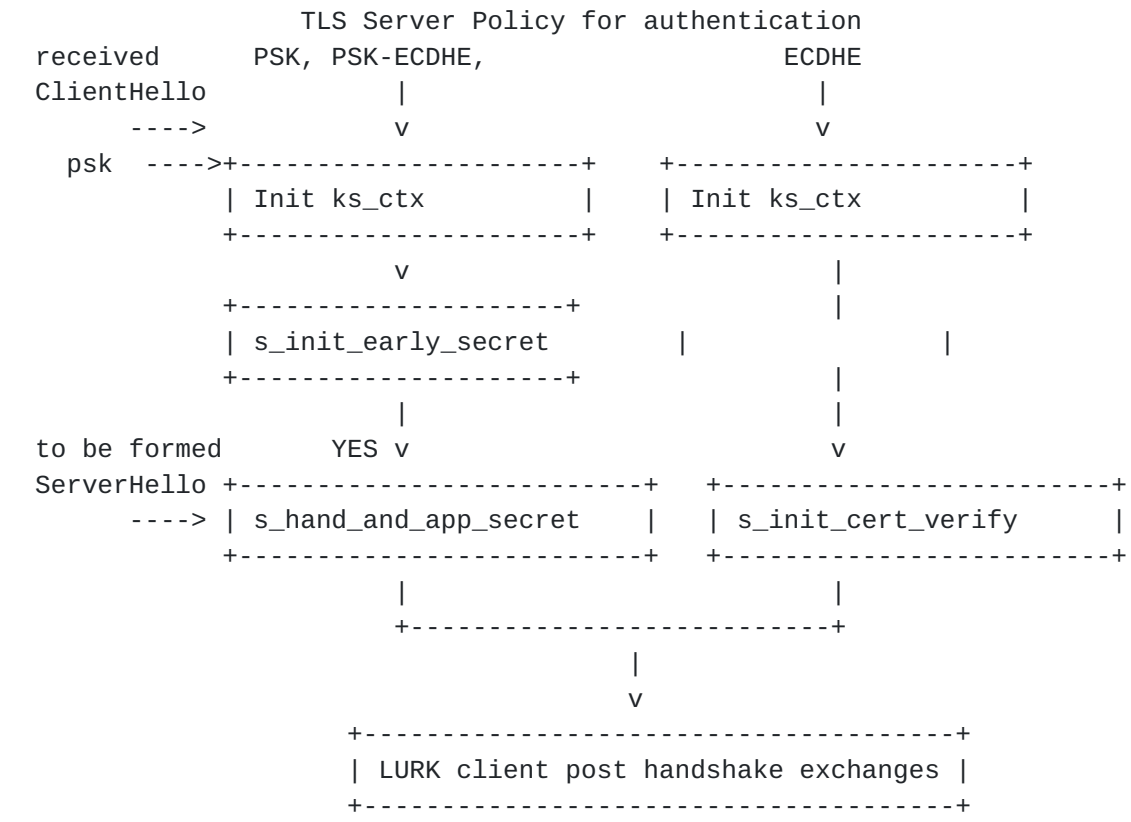


12.1.2. Cryptographic Service



12.2. LURK state diagrams on TLS server

12.2.1. LURK client



12.2.2. Cryptographic Service

12.3. TLS handshakes with Cryptographic Service

This section is non normative. It illustrates the use of LURK in various configurations.

The TLS client may propose multiple ways to authenticate the server (ECDHE, PSK or PSK-ECDHE). The TLS server may chose one of those, and this choice is reflected by the LURK client on the TLS server. In other words, this decision is out of scope of the CS.

The derivation of the secrets is detailed in {{!RFC8446)) section 7.1. Secrets are derived using Transcript-Hash and HKDF, PSK and ECDHE secrets as well as some Handshake Context.

The Hash function: When PSK or PSK-ECDHE authentication is selected, the Hash function is a parameter associated to the PSK. When ECDHE, the hash function is defined by the cipher suite algorithm negotiated. Such algorithm is defined in the cipher_suite extension provided in the ServerHello which is provided by the LURK client in the first request when ECDHE authentication is selected.

PSK secret: When PSK or PSK-ECDHE authentication is selected, the PSK is the PSK value identified by the identity. When ECDHE

authentication is selected, the PSK takes a default value of string of Hash.length bytes set to zeros.

ECDHE secret: When PSK or PSK-ECDHE authentication is selected, the ECDHE secret takes the default value of a string of Hash.length bytes set to zeros. The Hash is always known as a parameter associated to the selected PSK. When ECDHE authentication is selected, the ECDHE secret is generated from the secret key (ephemeral_sercet) provided by the LURK client and the counter part public key in the key_share extension. When the LURK client is on the TLS client, the public key is provided in the ServerHello. When the LURK client is on the TLS Server, the public key is provided in the ClientHello. When ECDHE secret is needed, ClientHello...ServerHello is always provided to the CS.

Handshake Context: is a subset of Handshake messages that are necessary to generated the requested secrets. The various Handshake Contexts are summarized below:

Key Schedule secret or key	Handshake Context
binder_key	None
client_early_traffic_secret	ClientHello
early_exporter_master_secret	ClientHello
client_handshake_traffic_secret	ClientHello...ServerHello
server_handshake_traffic_secret	ClientHello...ServerHello
client_application_traffic_secret_0	ClientHello...server Finished
server_application_traffic_secret_0	ClientHello...server Finished
exporter_master_secret	ClientHello...server Finished
resumption_master_secret	ClientHello...client Finished

The CS has always the Hash function, the PSK and ECDHE secrets and the only remaining parameter is the Handshake Context. The remaining sections will only focus on checking the Handshake Context available to the CS is sufficient to perform the key schedule.

When ECDHE authentication is selected both for the TLS server or the TLS client, a CertificateVerify structure is generated as described in [RFC8446] section 4.4.3.. CertificateVerify consists in a signature over a context that includes the output of Transcript-Hash(Handshake Context, Certificate) as well as a context string. Both Handshake Context and context string depends on the Mode which is set to server in this case via the configuration of the LURK server. Similarly to the key schedule, the Hash function is defined by the PSK or the ServerHello. The values for the Handshake Context are represented below:

Mode	Handshake Context	Base Key
Server	ClientHello ... later of EncryptedExtensions/ CertificateRequest	server_handshake_traffic_ secret
Client	ClientHello ... later of server Finished/EndOfEarlyData	client_handshake_traffic_ secret
Post-Handshake	ClientHello ... client Finished + CertificateRequest	client_application_traffic_ secret_N

When ECDHE authentication is selected, the CS generates a Finished message, which is a MAC over the value Transcript-Hash(Handshake Context, Certificate, CertificateVerify) using a MAC key derived from the Base Key. As a result, the same Base Key and Handshake Context are required for its computation describe din [RFC8466] section 4.4.4..

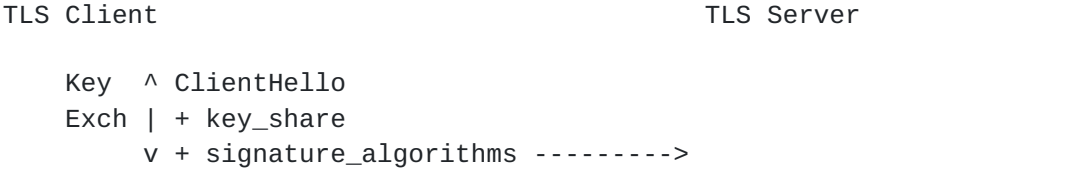
12.4. TLS 1.3 ECDHE Full Handshake

This example illustrates the case of a TLS handshake where the TLS server is authenticated using ECDHE only, that is not PSK or PSK-ECDHE authentication is provided and so session resumption is provided either.

12.4.1. TLS Client: ClientHello

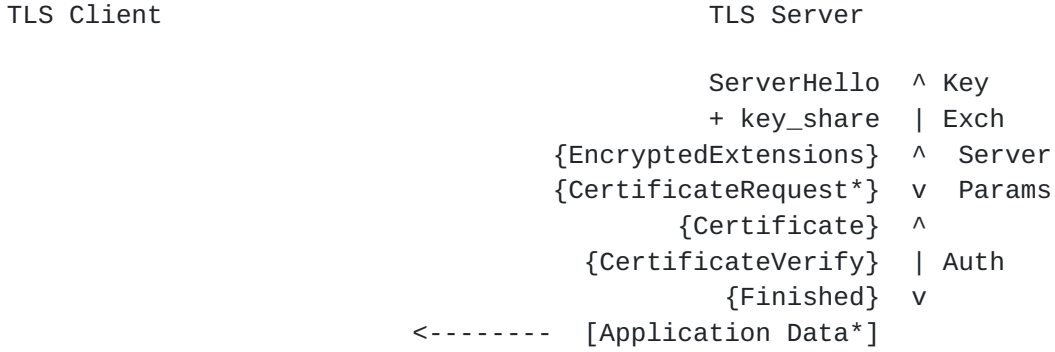
The TLS client does not provides any PSK and omits the pre_shared_key as well as the psk_key_exchange_mode extensions. Note that omitting the psk_key_exchange_mode extension prevents the TLS client to perform further session resumption.

The TLS client does not need any interaction with the Cryptographic Service to generate and send the ClientHello message to the TLS server.



12.4.2. TLS Server: ServerHello

Upon receiving the ClientHello, the TLS server determines the TLS client requests an ECDHE authentication. The TLS server initiates a LURK session to provide ECDHE authentication as represented below:



The LURK Client on the TLS server initiates a s_init_cert_verify to retrieves the necessary secrets to finish the exchange and request the generation of the signature (certificate_verify) carried by the CertificateVerify TLS structure.

The s_init_cert_verify request uses a InitCertVerifyRequest structure which is composed of two substructures: A SecretRequest structure (OLD_secret_request) is in charge of requesting the necessary secrets to decrypt and encrypt the TLS handshake as well as the applications carried over the TLS session. Finally a SigningRequest substructure (signing_request) is used to request the certificate_verify payload.

The OLD_secret_request carries the requested secrets as well as the necessary parameters to generate the secrets. In our case, the requested secrets are the handshake secrets (h_c, h_s) as well as the application secrets (a_c, a_s). This corresponds to the most expected use cases, though other use case may require different secrets to be requested. Theses requests are indicated in the secret_request. The necessary Handshake Context is provided through handshake which is set to ClientHello ... EncryptedExtensions. The ECDHE shared secret is provided in this example via the ephemeral extension. In our case, the secret key is provided directly thought other means may be used. In particularly providing the secret key implies the dhe parameters have been generated outside the CS. The freshness function is provided through the freshness extension.

The signing_request provides the key_id that identifies the private key used to generate the signature, the algorithm use dto generate the signature (sig_algo) as well as the certificate. The certificate carries information to generate the Certificate structure of the

ServerHello, and may not be the complete certificate chain but only an index.

Since there is no session resumption, the requests indicates with the tag set to last_exchange that no subsequent messages are expected. As a result, no session_id is provided. The freshness function is set to sha256, the handshake is constituted with the appropriated messages with a modified server_random to provide PFS. Th eCertificate message is also omitted from the handshake and is instead provided in the certificate structure using a finger_print. The requested secrets are handshake and application secrets, that is h_s, h_c, a_s, and a_c. The signature scheme is ed25519. With authentication based on certificates, there are two ways to generate the shared secrets that is used as an input to the derive the secrets. The ECDHE private key and shared secret may be generated by the CS as described in {cs_generated}. On the other hand the ECDHE private key and shared secret may be generated by the TLS server as described in {tls_Server_generated}

12.4.3. ecdhe generated on the CS (#cs_generated}

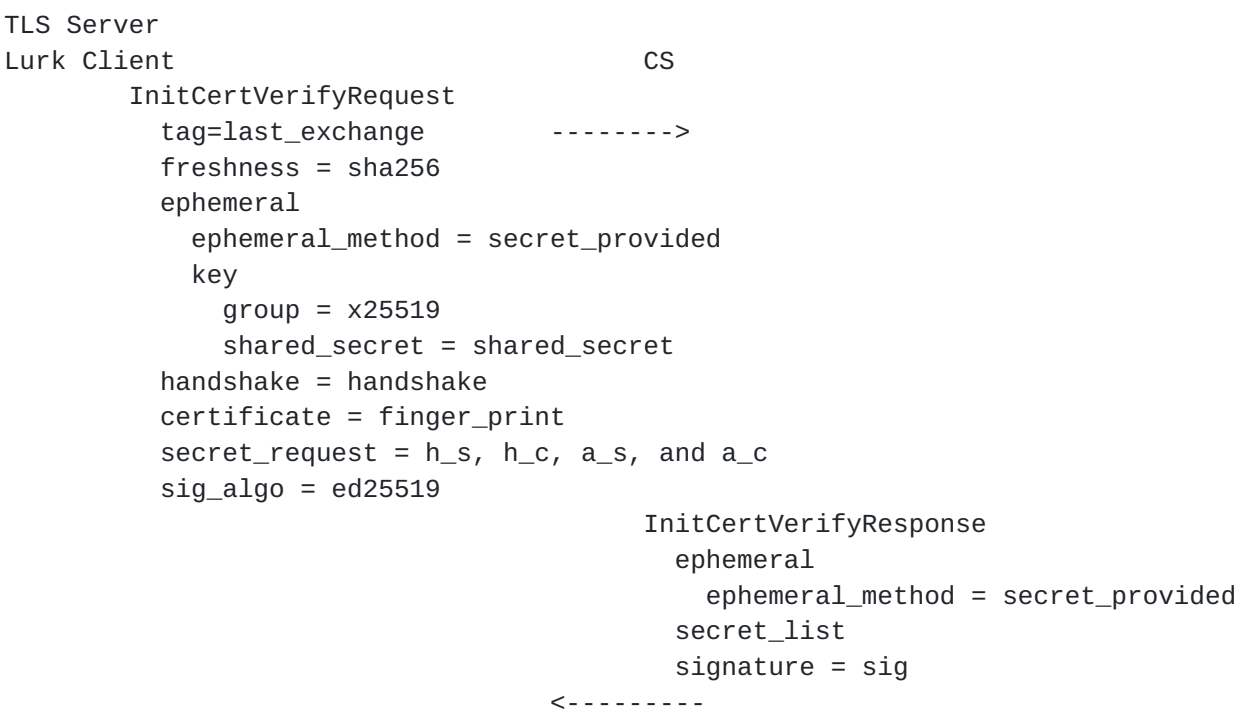
When the (EC)DHE private key and shared secrets are generated by the CS, the LURK client set the ephemeral_method to secret_generated. The (EC)DHE group x25519 is specified in the handshake in the key_share extension. In return the CS provides the LURK client the public key so the TLS server can send the ServerHello to the TLS client.

In this scenario, the CS is the only entity tthat knows the private ECDHE key and the shared secret, and only the CS is able to compute the secrets.

```
TLS Server
Lurk Client                                CS
        InitCertVerifyRequest
          tag=last_exchange        -------->
          freshness = sha256
          ephemeral
            ephemeral_method = secret_generated
          handshake = handshake (x25519)
          certificate = finger_print
          secret_request = h_s, h_c, a_s, and a_c
          sig_algo = ed25519
                                           InitCertVerifyResponse
                                             ephemeral
                                               ephemeral_method =
secret_generated
                                               key
                                                 group = x25519,
                                                 key_exchange = public_key
                                             secret_list
                                             signature = sig
                                    <---------
```

12.4.4. ecdhe generated by the TS server

When the (EC)DHE private keys and the shared secrets are generated by the TLS server, the LURK client provides the shared secret to the CS as only the shared secret is necessary to generated the signature. This is indicated by the ephemeral_method set to secret_provided. No (EC)DHE values are returned by the CS as these have already been generated by the TLS server. However, the TLS server has all the necessary material to generate the secrets and the only information that the CS owns and that is not known to et TLS server is the private key (associated to the certificate) used to generate the signature. This means that is session resumption were allowed, since it is based on PSK authentication derived from teh resumption secret, these session could be authenticated by the TLS server without any implication from the CS.



Upon receiving the InitCertificateRequest, the CS initiates a context associated to the newly created LURK session.

The secrets are generated from the TLS 1.3 key schedule describe din [RFC8446] and requires as input PSK, ECDHE as well as some context handshake.

The CS determine that ECDHE without specific PSK is used from the ClientHello and associated extensions. As a result, the default PSK value is used. The ECDHE share secret is derived, in our case from the dhe_secret of the TLS server and the public dhe value provided by the ClientHello shared_key extension.

The CS reads the freshness extension and generates the handshake that will be used further.

The necessary Handshake Context to generate the handshake secrets is ClientHello...ServerHello which is provided by the handshake. The CS uses the freshness function provided in the freshness extension to derive the appropriated server.random.

The generation of the CertificateVerify is described in [RFC8446] section 4.4.3. and consists in a signature over a context that includes the output of Transcript-Hash(Handshake Context, Certificate) as well as a context string. Both Handshake Context and context string depends on the Mode which is set to server in this case via the configuration of the LURK server.

The necessary Handshake Context to generate the CertificateVerify is ClientHello ... later of EncryptedExtensions / CertificateRequest. In our case, this is exactly handshake, that is ClientHello ... EncryptedExtensions. The Certificate payload is generated from the information provided in the certificate extension.

Once the certificate_verify value has been defined, the LURK server generates the server Finished message in order to have the necessary Handshake Context ClientHello...server Finished to generate the application secrets.

The LURK server returns the requested keys, the certificate_verify in a InitCertVerifyResponse structure. This structure is composed of the two substructures: SecretResponse that contains the secrets and SigningResponse that contains the certificate_verify.

The TLS server can complete the ServerHello response, that is proceed to the encryption and generates the Finished message.

As session resumption is not provided, the LURK server goes into a finished state and delete the ks_ctx. The special case described in this session does not use LURK session and as such may be stateless.

12.4.5. TLS client: client Finished

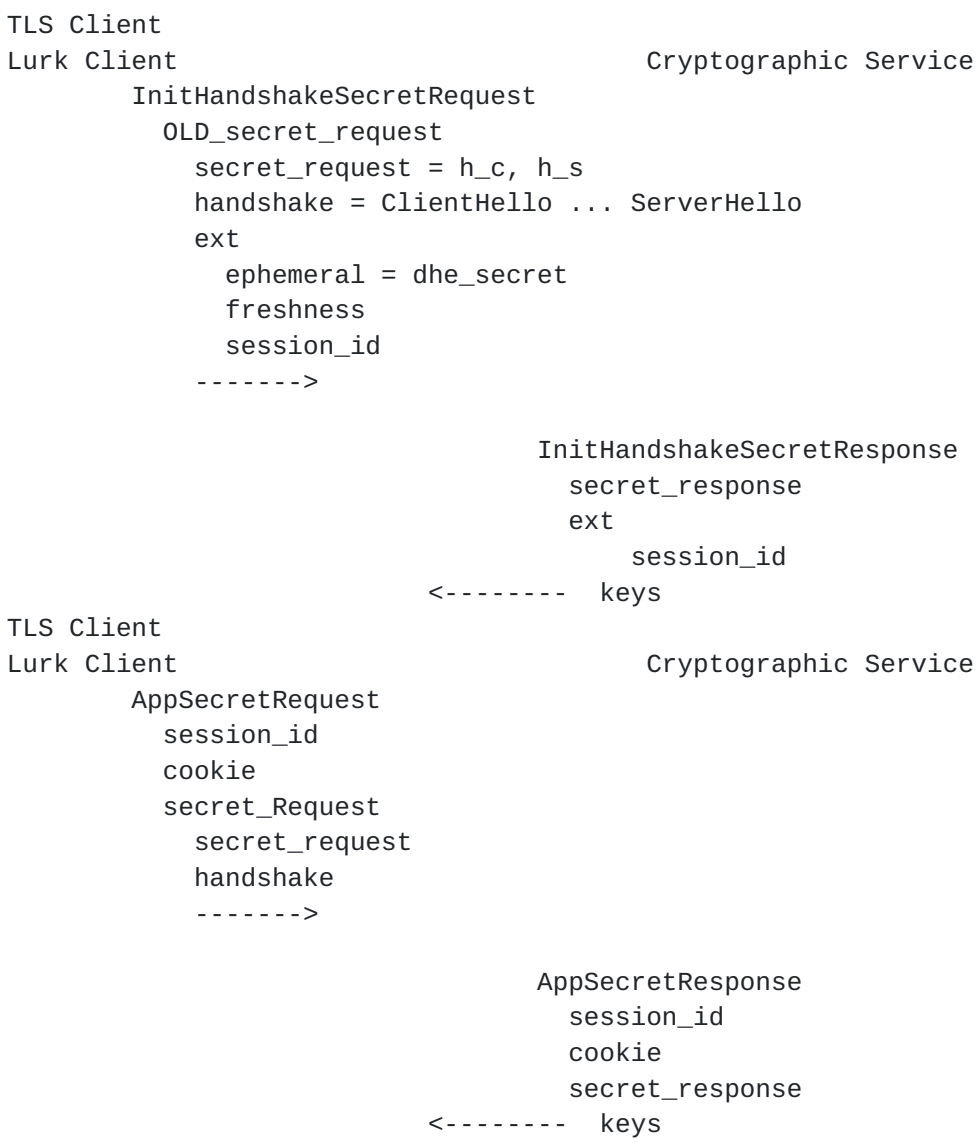
Upon receiving the ServerHello message, the TLS client retrieve the handshake and application secrets to decrypt the messages received from server as well as to encrypt its own messages and application data as represented below:

To retrieves these secrets, the TLS client proceeds successively to an c_init_hand_secret LURK exchange followed by a c_app_secret LURK exchange.

The c_init_hand_secret exchange is composed of one substructure: (OLD_secret_request) to request the secrets. Optionally, a SigningRequest (signing_request) when the TLS server requests the TLS client to authenticate itself. The indication of a request for TLS client authentication is performed by the TLS server by providing a CertificateRequest message associated to the ServerHello. We consider that such request has not been provided here so the SigingRequest structure is not present.

The OLD_secret_request specifies the secrets requested via the secret_request. In our case only the handshake secrets are requested (h_c, h_s). In this example the ECDHE share secret is provided via the ephemeral extension. In this case the ECDHE secrets have been generated by the TLS client, and the TLS client chooses to provide the ephemeral secret (dhe_secret) to the CS via the ephemeral extension. The TLS client also provides the freshness function via the freshness extension so the handshake can be appropriately be interpreted. The handshake context is provided via the handshake and is set to ClientHello ... ServerHello.

Note that if the TLS client would have like the CS to generate the ECDHE public and private keys, the generation of the keys would have been made before the ClientHello is sent, that is in our case during a c_init_early_secret LURK exchange. If that had been the case a c_hand_secret LURK exchange would have followed and not a c_init_hand_secret exchange.



Upon receiving the InitHandshakeSecretRequest, the servers initiates a LURK session context (ks_ctx) and initiates a key schedule. The key schedule requires PSK, ECDHE as well as Handshake Context to be complete. As no pre_shared_key and psk_key exchange_modes are found in the ClientHello the CS determines that ECDHE is used for the authentication. The PSK is set to its default value. The ECHDE shared secret is generated from the ephemeral extension as well as the public value provided in the ClientHello. The CS takes the freshness function and generates the appropriated handshake context. The necessary Handshake Context to generate handshake secrets is ClientHello...ServerHello which is provided by the handshake.

The handshake secrets are returned in the secret_response to the TLS client. The TLS client decrypt the encrypted extensions and messages of the ServerHello exchange.

As no CertificateREquest appears, the LURK client initiates an app_secret LURK exchange decrypt and encrypt application data while finishing the TLS handshake.

The AppSecretRequest structure uses session_id and cookies as agreed in the previous c_init_hand_secret exchange. The AppSecretRequest embeds a SecretRequest sub structure. The application secrets requested are indicated by the secret_request (a_s, a_s). The Handshake Context (handshake) is set to server EncryptedExtensions ... server Finished.

Upon receiving the AppSecretRequest, the CS checks the session_id. The CS has now the ClientHello ... server Finished which enables it to compute the application secrets.

As no session resumption is provided, the CS and the LURK client goes into a finished state and delete their ks_ctx.

12.5. TLS 1.3 Handshake with session resumption

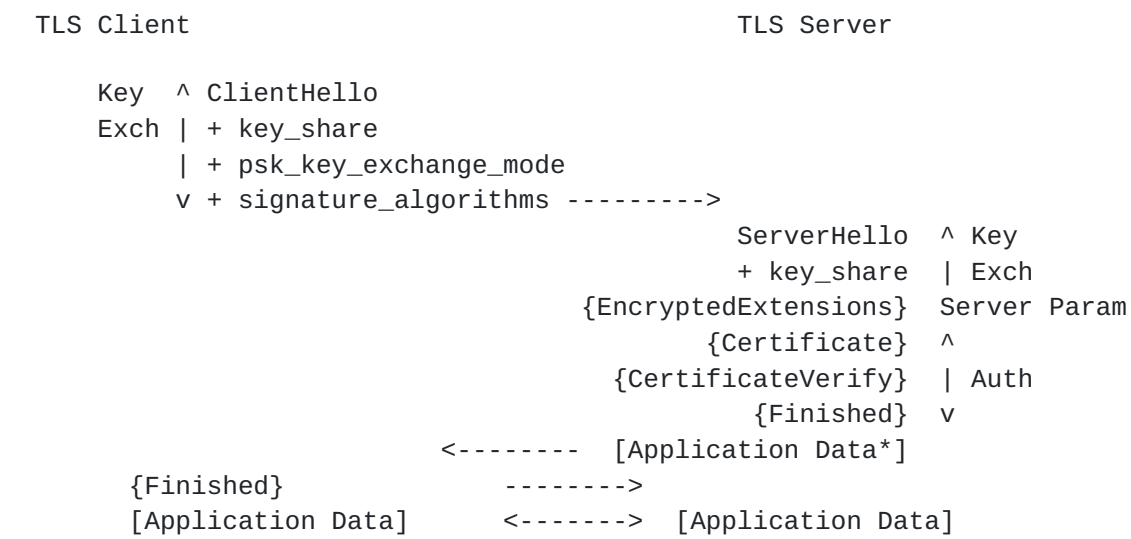
This scenario considers that the TLS server is authenticated using ECDHE only in the first time and that further TLS handshake use the session resumption mechanism. The first TLS Handshake is very similar as the previous one. The only difference is that psk_key_exchange_mode extension is added to the ClientHello. However, as no PSK identity is provided, the Full exchange is performed as described in section Section 12.4.

The only change is that session resumption is activated, and thus LURK client and LURK servers do not go in a finished state and close the LURK session after the exchanges are completed. Instead further exchanges are expected. Typically, on the TLS server side new_Session_ticket exchanges are expected while registered_session_ticket are expected on the client side.

When session resumption is performed, a new LURK session is initiated.

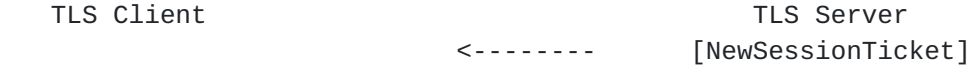
12.5.1. Full Handshake

The Full TLS Handshake use ECDHE authentication. It is very similar to the logic described in section Section 12.4. The TLS handshake is specified below for convenience.

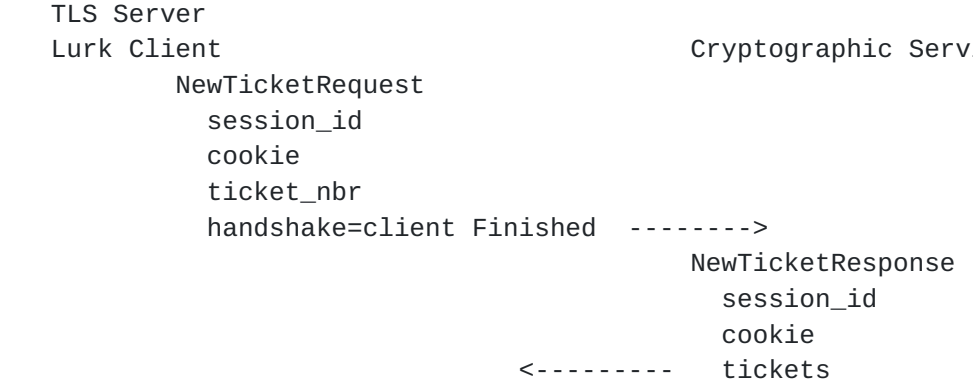


12.5.2. TLS server: NewSessionTicket

As session resumption has been activated by the psk_key_exchange_mode, the TLS Server is expected to provide the TLS client NewSessionTickets as mentioned below:



The LURK client and LURK server on the TLS server does not go into a finished state. Instead, the LURK client continues the LURK session with a NewTicketRequest to enable the CS to generate the resumption_master_secret necessary to generate the PSK and generate a NewTicketSession. ticket_nbr indicates the number of NewSessionTickets and handshake is set to earlier of client Certificate client CertificateVerify ... client Finished. As we do not consider TLS client authentication, the handshake is set to client Finished as represented below.



The necessary Handshake Context to generate the resumption_master_secret is ClientHello...client Finished. From the InitCerificateVerify the context_handshake was set to ClientHello...server Finished. The additional handshake enables the CS to generate the NewSessionTickets.

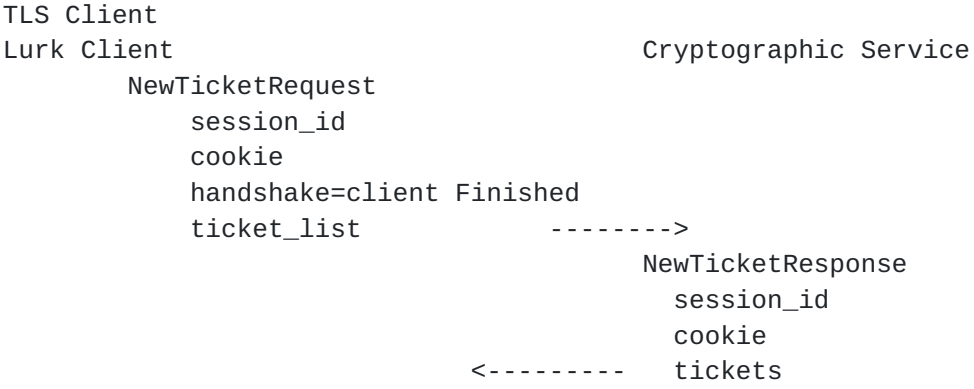
Note that the LURK client on the TLS server may send multiple NewTicketRequest. Future request have an empty handshake.

Upon receiving the NewTicketRequest, the LURK server checks the session_id and cookie. It then generates the resumption_master_secret, NewSessionTickets. NewSessionTickets are stored into the PSK_DB under NewSessionTicket.ticket. Note that PSK is associated with the authentication mode as well as the Hash function negotiated for the cipher suite. The CS responds with NewSessionTickets that are then transmitted back to the TLS client. The TLS server is ready for session resumption.

12.5.3. TLS client: NewSessionTicket

Similarly, the LURK client on the TLS client will have to provide sufficient information to the CS the necessary PSK can be generated in case of session resumption. This includes the remaining Handshake Context to generate the resumption_master_secret as well as NewSessionTickets provided by the TLS server. The LURK client uses the c_register_ticket exchange.

Note that the LURK client may provide the handshake with an empty list of NewSessionTickets, and later provide the NewSessionTickets as they are provided by the TLS server. The Handshake Context only needs to be provided for the first RegisterTicketRequest.

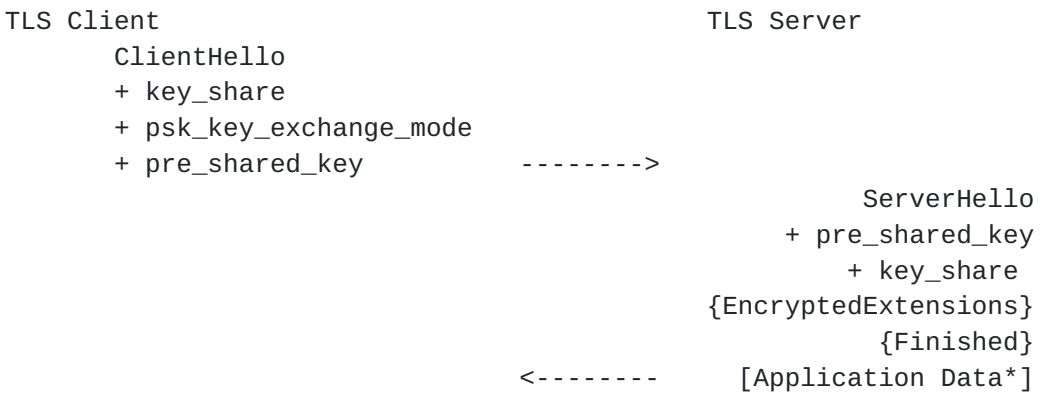


Both TLS client and TLS Servers are ready for further session resumption. On both side the CS stores the PSK in a database designated as PSK_DB. Each PSK is associated to a Hash function as well as authentication modes. Each PSK is designated by an identity.

The identity may be a label, but in our case the identity is derived from the NewSessionTicket.ticket.

12.5.4. Session Resumption

Session resumption is initiated by the TLS client. Session resumption is based on PSK authentication and different PSK may be proposed by the TLS client. The TLS handshake is presented below.



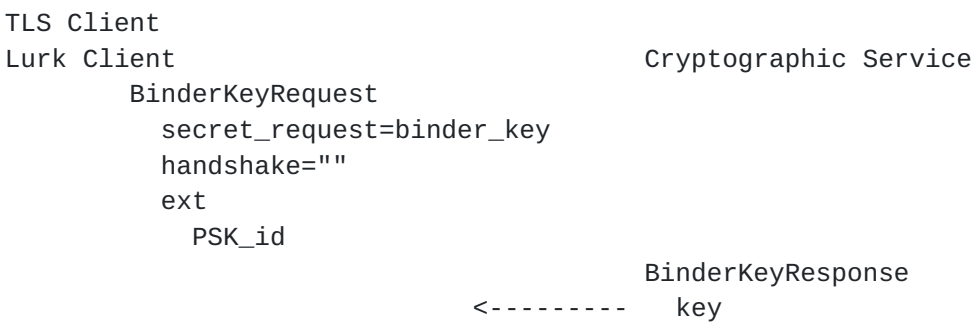
The TLS client may propose to the TLS Server multiple PSKs. Each of these PSKs is associated a PskBindersEntry defined in [RFC8446] section 4.2.11.2. PskBindersEntry is computed similarly to the Finished message using the binder_key and the partial ClientHello.

The TLS server is expected to pick a single PSK and validate the binder. In case the binder does not validate the TLS Handshake is aborted. As a result, only one binder_key is expected to be requested by the TLS server as opposed to the TLS client.

In this example we assume the psk_key_exchange_mode indicated by the TLS client supports PSK-ECDHE as well as PSK authentication. The presence of a pre_shared_key and a key_share extension in the ServerHello inidcates that PSK-ECDHE has been selected.

12.5.4.1. TLS client: ClientHello

To compute binders, the TLS Client needs to request the binder_key associated to each proposed PSK. These binder_keys are retrieved to the CS using the BinderKeyRequest. The secret_request is set to binder_key, and the PSK_id extension indicates the PSK's identity (PSKIdentity.identity or NewSessionTicket.ticket). No Handsahke Context is needed and handshake is empty.



Upon receiving the BinderKeyRequest, the CS checks the psk is in the PSK_DB and returns the binder_key.

With the binder keys, the TLS Client is able to send it ClientHello message.

We assume in this example that the ECDHE secrets is generated by the TLS client and not the Cryptographic service. As a result, the TLS client does not need an extra exchange to request the necessary parameters to derive the key_shared extension.

12.5.4.2. TLS server: ServerHello

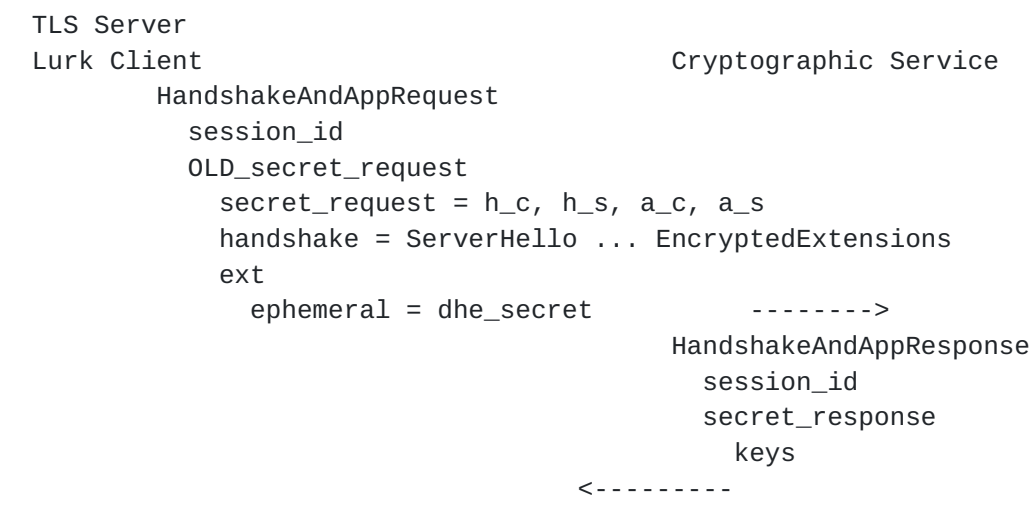
The TLS server is expected to select a PSK, check the associated binder and proceed further. If the binder fails, it is not expected to proceed to another PSK, as a result, the TLS server is expected to initiates a single LURK session.

The binder_key is requested by the TLS server via and s_init_early_secret LURK exchange. The InitEarlySecretRequest structure is composed of a SecretRequest structure (OLD_secret_request).

In our case, only the binder_key is requested so secret_request is set to binder_key only. Similarly, to the TLS client, the handshake is not needed to generate the binder_key. However, the EarlySecret exchange requires the ClientHello to be provided so early secrets may be computed in the same round during 0-RTT handshake. The chosen PSK is indicated in the PSK_id extension and the freshness function is indicated in the freshness extension.

To complete to the ServerHello exchange, the TLS server needs the handshake and application secrets. These secrets are requested via an s_hand_and_app_secret LURK exchange. The HandshakeAndAppSecretRequest is composed of SecretRequest structure. The secret_request is set to handshake (h_c, h_s) and application secrets (a_s, a_c). The Handshake Context (handshake) is set to ServerHello ... EncryptedExtensions as their is no authentication of the TLS client. Finally, the ephemeral ECDHE is provided or requested via the ephemeral extension. In our case, we assume the ephemeral secrets is generated by the tLS client is provided to the CS.

The necessary Handshake Context to generate the handshake secrets is ClientHello ... ServerHello, so the CS can generate the handshake secrets. The necessary Handshake Context to generate the application secrets is ClientHello ... server Finished. So the CS needs to generate the Finished message before as in the case of the InitCerificateVerify exchange detailed in Section 12.5.1.



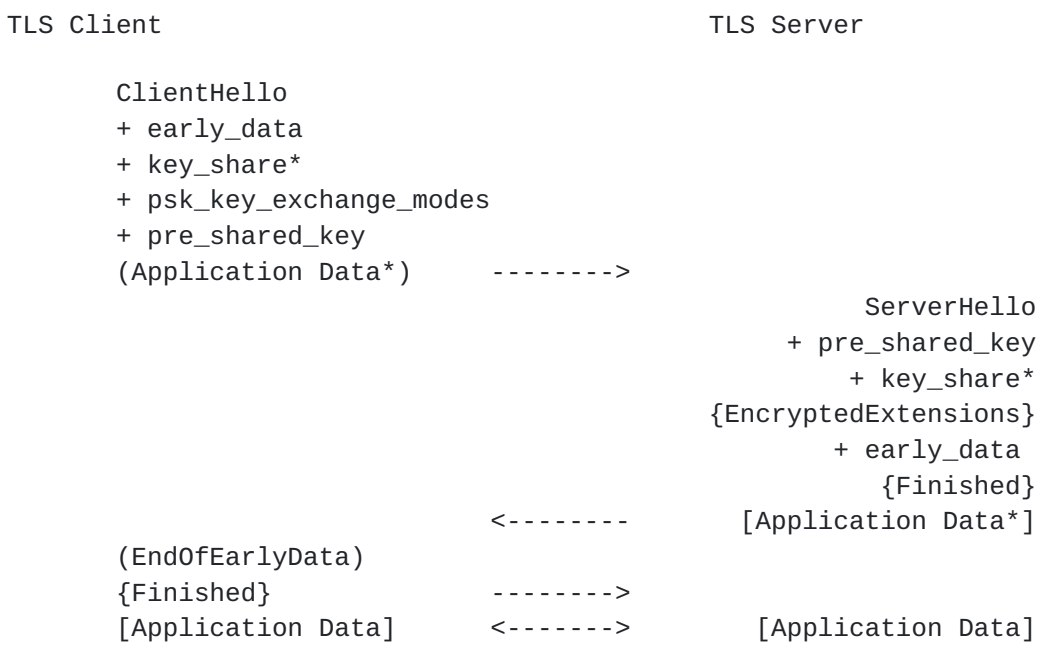
The CS returns the necessary secret to the TLS server to complete the ServerHello response.

The remaining of the TLS handshake is proceeded similarly as described in the Full Handshake in section Section 12.5.

12.6. TLS 1.3 0-RTT handshake

The 0-RTT Handshake is a PSK or PSK-ECDHE authentication that enables the TLS client to provide application data during the first round trip. The main differences to the PSK PSK-ECDHE authentication described in the case of session resumption is that:

- Application Data is encrypted in the ClientHello based on the client_early_secret
- Generation of the client_early_secret requires the Cryptographic Service to be provisioned with the ClientHello which does not need to be re-provisioned later to generate the handshake secrets
- An additional message EndOfEarlyData needs to be considered to compute the client Finished message.



12.6.1. TLS client: ClientHello

With 0-RTT handshake, the TLS client builds binders as in session resumption described in section Section 12.5.4. The binder_key is retrieved for each proposed PSK with a BinderKeyRequest. When early application data is sent it is encrypted using the client_early_traffic_secret. This secret is retrieved using the c_init_early_secret LURK exchange.

The InitEarlySecretRequest is composed of a SecretRequest (OLD_secret_request) substructure. The TLS Client sets the secret_request to client_early_traffic_secret (e_s). The handshake is set to ClientHello. The PSK is indicated via the the PSK_id extension, the freshness function is indicated via the freshness extension. If the TLS client is willing to have the ECDHE keys generated by the CS an ephemeral extension MAY be added also.

When multiple PSK are proposed by the TLS client, the first proposed PSK is used to encrypt the application data.

Upon receiving the InitEarlySecretRequest, the CS generates the client_early_traffic_secret.

The TLS client is able to send its ClientHello with associated binders and application data.

12.6.2. TLS server: ServerHello

If the TLS server accepts the early data. It proceeds as described in session resumption described in section Section 12.5.4. In addition to the binder_key, the TLS server also request the client_early_traffic_secret to decrypt the early data as well as to proceed to the ServerHello exchange.

12.6.3. TLS client: Finished

The TLS client proceeds as described in handshake based on ECDHE, PSK or PSK-ECDHE authentications described in Section 12.4 and Section 12.5. The main difference is that upon requesting handshake and application secrets, using an HandAndAppRequest the TLS client will not provide the ClientHello as part as the handshake. The Client as already been provided during the EarlySercret exchange.

12.7. TLS client authentication

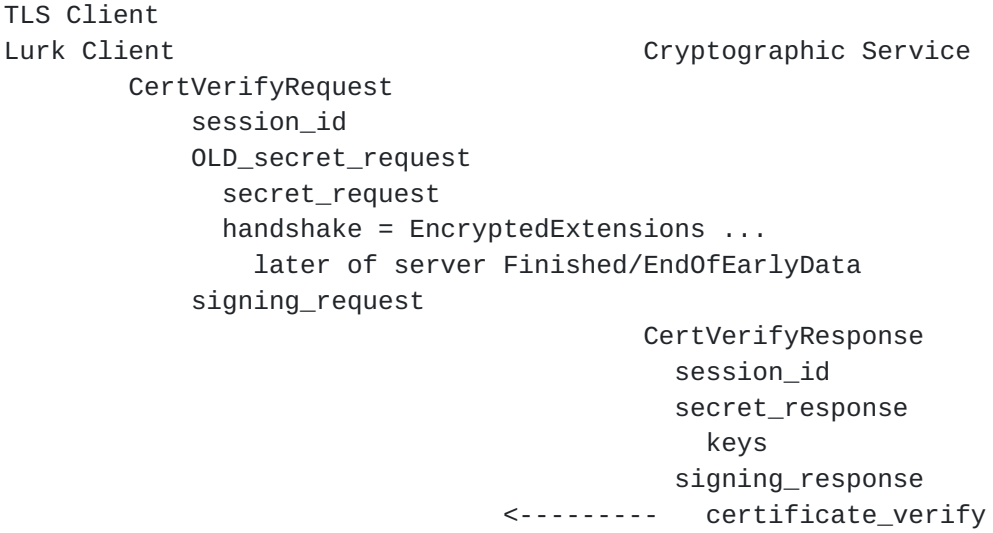
TLS client authentication can be performed during the Full TLS handshake or after the TLS handshake as a post handshake authentication. In both cases, the TLS client authentication is initiated by the TLS server sending a CertificateRequest. The authentication is performed via a CertificateVerify message generated by the TLS client but such verification does not involve the CS on the TLS server.

12.8. TLS Client:Finished (CertificateRequest)

The ServerHello MAY carry a CertificateRequest encrypted with the handshake sercets.

Upon receiving the ServerHello response, the TLS client decrypts the ServerHello response. If a CertificateRequest message is found, the TLS Client requests the Cryptographic to compute the CertificateVerify in addition to the application secrets via a certificate_verify LURK exchange. The CertVerifyRequest is composed of a Secret Request structure and a SigningRequest structure.

The secret_request is set to the application secrets (a_c, a_s) and the handshake is set to server EncryptedExtensions ... later of server Finished/EndOfEarlyData. As the request follows a (BinderKey, EarlySecret, HandshakeSecret) or HandshakeSecret the Handshake Context on the CS now becomes: ClientHello ... later of server Finished/EndOfEarlyData which is the Handshake Context required to generate the CertificateVerify on the TLS client side and includes the Handshake Context required to generate the application secrets (ClientHello...server Finished).



Upon receiving the CertificateRequest, the CS checks the session_id and cookie.

12.9. TLS Client Authentication (PostHandshake)

When post-handshake is enabled by the TLS client, the TLS client may receive at any time after the handshake a CertificateRequest message. When post handshake is enabled by the TLS client, as soon as the client Finished message has been sent, the TLS client sends a RegisteredNewSessionTicketRequest with an empty NewSessionTicket to

register the remaining Handshake Context to the CS. ctx_id is set to opaque, handshake is set to earlier of client Certificate client CertificateVerify ... client Finished.

Upon receiving the RegisteredNewSessionTicketsRequest the Cryptographic is aware of the full Handshake Context. It updates ks_ctx.next_request to c_post_hand or c_register_ticket.

```
TLS Client
Lurk Client                                Cryptographic Service
        RegisteredNewSessionTicketRequest
            session_id
            handshake
            ticket_list (empty)
                  <--------- RegisteredNewSessionTicketResponse
                               session_id
                               cookie
```

When the TLS client receives a CertificateRequest message from the TLS server, the TLS client sends a PostHandshakeRequest to the Cryptographic Service to generate certificate_verify. The handshake is set to CertificateRequest. The index N of the client_application_traffic_N key is provided as well as the Cryptographic so it can generate the appropriated key.

```
TLS Client
Lurk Client                                Cryptographic Service
        PostHandshakeRequest
          session_id
          handshake=CertificateRequest
          app_n=N
                                           PostHandshakeResponse
                                             session_id
                              <---------     certificate_verify
```

Upon receiving the PostHandshakeRequest the CS checks session_id and cookie. The necessary Handshake Context to generate the certificate_verify is ClientHello ... client Finished + CertificateRequest. Once the PostHandshakeResponse. Next requests expected are c_post_hand or c_register_ticket.

13. References

13.1. Normative References

[I-D.ietf-tls-certificate-compression]
Ghedini, A. and V. Vasiliev, "TLS Certificate Compression", draft-ietf-tls-certificate-compression-10 (work in progress), January 2020.

[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>.

[RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, July 2016, <https://www.rfc-editor.org/info/rfc7924>.

[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>.

[RFC8466] Wen, B., Fioccola, G., Ed., Xie, C., and L. Jalil, "A YANG Data Model for Layer 2 Virtual Private Network (L2VPN) Service Delivery", RFC 8466, DOI 10.17487/RFC8466, October 2018, <https://www.rfc-editor.org/info/rfc8466>.

13.2. Informative References

[I-D.mglt-lurk-lurk]
Migault, D., "LURK Protocol version 1", draft-mglt-lurk-lurk-00 (work in progress), February 2018.

[I-D.mglt-lurk-tls12]
Migault, D. and I. Boureanu, "LURK Extension version 1 for (D)TLS 1.2 Authentication", draft-mglt-lurk-tls12-03 (work in progress), July 2020.

Author's Address

Daniel Migault
Ericsson
8275 Trans Canada Route
Saint Laurent, QC 4S 0B6
Canada

EMail: daniel.migault@ericsson.com