Workgroup: LURK
Internet-Draft: draft-mglt-lurk-tls13-06
Published: 24 August 2022
Intended Status: Standards Track
Expires: 25 February 2023
Authors: D. Migault
Ericsson

LURK Extension version 1 for (D)TLS 1.3 Authentication

Abstract

This document defines a LURK extension for TLS 1.3 [RFC8446], with the specification of a Cryptographic Service (CS) for both the TLS client and the TLS server.

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 25 February 2023.

Copyright Notice

Table of Contents

1. Introduction

This document defines the LURK extension for TLS 1.3 The document considers the Private Key (PK) used generate signature of CertificateVerify message is always protected by the CS. Additionally PSK or the (EC)DHE secret key MAY also be protected by the CS.[RFC8446] also designated as 'tls13'. This extension enables TLS 1.3 to be securely split between TLS 1.3 into a TLS Engine (E) and a Cryptographic Service (CS) for both the TLS client and the TLS server.

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

E interacts with a CS to perform three types of operations: perform a signature with a secret private key, generate secrets from the TLS 1.3 key schedule or generate tickets for future sessions. To limit the number of exchanges between E and CS, packs these operations across various possible LURK exchanges as summed up in Table Figure 1. As a result, these exchanges do share many common structures, each exchange happens in a very specific state with a specific subset of structures that results in such exchange being uniquely defined.

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_init_client_finished	-	yes	-
client	c_post_hand_auth	-	yes	-
client	c_init_client_hello	yes	-	-
client	c_server_hello	yes	-	-
client	c_client_finished	yes	yes	-
client	c_register_tickets	yes	-	yes

Figure 1: Operation associated to LURK exchange

2. 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]. However, we replaced [RFC8446] (EC)DHE authentication by certificate based authentication to make a clear distinction between the generation of the signature, and the generation of (EC)DHE which may only be involved in PSK based authentication.

Private Key (PK) designates the private key associated to the certificate. PK is used to generate the signature of the CertificateVerify message when the TLS client or TLS server use certificate base authentication.

3. LURK Header

The LURK Extension described in this document is designated by a designation set to 'tls13' and a 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),  // to be removed
   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_tickets(12),
   c_post_hand_auth(13), (255)
}TLS13Type;


enum {
    // generic values reserved or aligned with the
    // LURK Protocol
    request (0), success (1),
    invalid_extention // to be added in lurk lurk
    undefined_error (2),
    invalid_format (3),
    invalid_type  // to be added in lurk_lurk
    invalid_status // to be added in lurk_lurk

    invalid_secret_request // to be removed
    invalid_session_id
    invalid_handshake
    invalid_freshness
    invalid_ephemeral
    invalid_psk
    invalid_certificate
    invalid_cert_type

    ///not in the code
    invalid_key_id_type
    invalid_signature_scheme
    invalid_certificate_type
    invalid_certificate_verify
    invalid_identity
    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;
```

4. Structures

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

4.1. secret_request

secret_request is a 16 bit structure described in Table Figure 2 that indicates the requested key or secrets. The same structure is used across multiple exchanges, but each exchange only permit a subset of values described in Table Figure 3. For a given exchange, values or secrets that are not permitted MUST NOT be requested by E and MUST be ignored by the CS. The secret request sent by E expresses a willingness for a given set of secrets. CS SHOULD provide the requested secrets, its response may implement specific policies and CS MAY omit some requested permitted secrets as well as add some permitted secrets.

Note that for the c_init_client_hello, CS MUST provide binder_keys the binder_key_list and MUST omit binder_key in the secret_list (see Section 6.3).

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_new_ticket	r
s_init_early_secret	b,e_c, e_x
s_hand_and_app_secret	h_c, h_s, a_c, a_s, x
c_init_client_finished	-
c_post_hand_auth	-
c_init_client_hello	e_c, e_x
c_server_hello	h_c, h_s,
c_client_finished	a_c, a_s, x, r
c_register_tickets	-

Figure 3: secret_request permitted values per LURK exchange

4.2. handshake

The derivation of the secrets, signing operation and tickets requires the TLS handshake context as described in [RFC8446] section 4. The necessary TLS handshake context is collected by CS in many ways. Firstly, E provides portions of the handshake during the multiple exchanges that is aggregated by the CS. Table Figure 4 shows the expected portion of the handshake transmitted by E to CS for the various possible exchanges. Secondly, CS generates some portions or update portions of the transmitted handshake provided by E.

Note that Certificate are not carried via the handshake structure but the cert structure as detailled in Section 4.7.

The handshake structure is defined as follows:

Handshake handshake<0..2^32> //RFC8446 section 4 (clear)

Table Figure 4 defines the content of the handshake parameter in the various exchanges.

LURK exchange	handshake content
s_init_cert_verify	ClientHello ... later of server EncryptedExtensions / CertificateRequest
s_new_ticket	earlier of client CertificateVerify / Finished ... Finished
s_init_early_secret	ClientHello
s_hand_and_app_secret	ServerHello ... later of server EncryptedExtensions / CertificateRequest
c_init_client_finished	ClientHello... later of server Finished/ EndOfEarlyData
c_post_hand_auth	CertificateRequest
c_init_client_hello	PartialClientHello or ClientHello HelloRetryRequest, PartialClientHello
c_server_hello	ServerHello
c_client_finished	EncryptedExtensions ... later of server Finished, EndOfEarlyData OR ServerHello, EncryptedExtensions ... later of server Finished, EndOfEarlyData
c_register_tickets	-

Figure 4: handshake values per LURK exchange

Upon receiving a handshake parameter, CS performs some checks described below:

psk_proposed A TLS handshake is "psk_proposed" when the TLS client proposes a PSK authentication. The latest ClientHello contains a psk_key_exchange_modes (section 4.2.9 of [RFC8446]) and a pre_shared_key (section 4.2.11 of [RFC8446]) extension.

psk_agreed A "psk_proposed" TLS handshake is "psk_agreed" when the TLS server agrees and selects PSK to authenticate to be authenticated by the TLS client. The ServerHello contains a pre_shared_key extension as according to [RFC8448] section 4.2.9 the Server MUST NOT send a psk_key_exchange_modes extension.

certificate_agreed As currently defined, TLS 1.3 [RFC8446] provides only PSK or certificate authentication. In addition, when the TLS server is authenticated with a certificate, according to [RFC8446] section 4.2.3), the ClientHello MUST contain a signature_algorithms extension.

ks_proposed A TLS handshake is "key shared proposed" when the computation of a (EC)DHE shared secret is proposed by the TLS

client. The latest ClientHello contains a key_share extension (section 4.2.8 of [RFC8446]).

ks_agreed A TLS handshake is key shared agreed or "ks agreed" when the TLS server agrees and complete the generation of a (EC)DHE shared secret. The ServerHello contains a key_shared extension.

certificate_request A TLS handshake is "certificate request" when the TLS server has requested the client to authenticate via a certificate by sending a CertificateRequest message with a signature algorithm extension.

early_data_proposed A TLS handshake is "early_data_proposed" when it has indicates the presence of early data. Its latest ClientHello contains a early_data extension (section 4.2.10 of [RFC8446].

early_data_agreed A TLS handshake is "early_data_agree" when a early_data extension is present in the Encrypted Extension.

post_hand_auth_proposed: A TLS handshake is "post_hand_auth_proposed" when the TLS client indicates the support of Post-Handshake Client Authentication. The ClientHello contains a post_handshake_auth extension.

4.2.1. s_init_cert_verify

This exchange implies the authenticated mode is (EC)DHE and CS MUST check the handshake is not 'psk_agreed' and is both 'ks_proposed' and 'ks_agreed'. CS SHOULD raise an invalid_handshake error otherwise.

CS applies the freshness function to the ServerHello.random as detailed in Section 4.4. When instructed to, CS generates the (EC)DHE and add it to its local ServerHello as detailed in Section 4.5.

The TLS server Certificate message is provided via the certificate structure detailed CS generates the Certificate message from the certificate structure. CS SHOULD raise an invalid_handshake_error if the server Certificate message is in found in the handshake structure.Section 4.7 and not directly in the handshake structure. In fact, the server Certificate message is rather considered as a configuration whose transmission for each handshake is loading unnecessarily the CS.

In addition to the server Certificate message, the CertificateVerify as described in Section 4.10 as well as the server Finished message as detailed in [RFC8446] section 4.4.4 to avoid a potential additional interaction between E and CS.

4.2.2. s_new_ticket

When the TLS handshake is "certificate request" the TLS handshake is expected to have a Certificate message provided by the certificate structure detailed Section 4.7 and not directly in the handshake structure.

4.2.3. s_init_early_secret

This exchange is expected to occur when a PSK has been selected and CS MUST check the handshake is 'psk_proposed' and SHOULD raise an invalid_handshake error otherwise.

CS applies the freshness function to ServerHello.random as detailed in Section 4.4.

CS MUST check the presence of an early_data extension in the ClientHello before generating the client_early_traffic_secret (e_x). If the extension is not found CS SHOULD NOT compute the secret. Note that the secret may be generated by CS without knowing E actually agreed on using it.

4.2.4. s_hand_and_app_secret

This exchange is expected to occur when a PSK has been selected and CS MUST check the handshake is 'psk_agreed' and SHOULD raise an invalid_handshake error otherwise. CS MUST check the ticket selected in the ServerHello is the same ticket as the one selected in the 's_init_early_secret'. To do so, CS MUST check SInitEarlySecret.selected_identity equals selected identity of the ServerHello pre_shared_key extension and SHOULD raise an invalid_handshake error otherwise. Similarly, CS MUST check the selected selected cipher suite has the same KDF hash algorithm as that used to establish the original connection - as per {!RFC8446} Section 4.6.1. To do so, CS MUST check the cipher suite selected by the KDF hash algorithm associated to the ServerHello message corresponds to the one provided by the ticket and SHOULD raise an invalid_handshake error otherwise.

When instructed to, CS generates the (EC)DHE and add it to its local ServerHello as detailed in Section 4.5. CS also generates the server finished message as detailed in [RFC8446] section 4.4.4.

4.2.5. c_init_client_finished

This exchange occurs when the TLS client has proposed client post handshake authentication or when the TLS server is requesting client authentication during the TLS handshake. As a result, CS MUST check the handshake is post_hand_auth_proposed or certificate_request and raise an 'invalid_handshake' error otherwise.

Note that the Handshake context MAY but not necessarily carry a server Certificate message or a client Certificate message. These message are not part of the handshake and carried in Cert structures as defined in Section 4.7.

4.2.6. c_post_hand_auth

This exchange occurs when the TLS client is authenticated via its certificate after the main handshake.

The CS MUST check E has indicated support of the post handshake client authentication. This is enforced by checking the TLS handshake is post_hand_auth_proposed. c_post_hand_auth handshake is composed of the CertificateRequest message.

4.2.7. c_init_client_hello

The c_init_client_hello message carries a partial client hello that corresponds to a ClientHello message to which the pre_shared_key and key_share extensions have been modified.

When the TLS client proposes PSK based authentication, E indicates the supported PSK via the PreSharedKeyExtension at the end of the ClientHello. As described in [RFC8446], the PreSharedKeyExtension contains an OfferedPsks structure that contains identities (a list of PskIdentity structure) and binders (a list of PskBinderEntry structure). As the PskBinder entries requires the binder_key to be generated and that binder_key is request with the c_init_client_hello at least for PSKs protected by CS, E provides a partial ClientHello where OfferedPsks.binders is omitted. The OfferedPsks structure is thus replaced OfferedPsksWithNoBinders as detailed below:

```
struct{
    PskIdentity identities<7..2^16-1>;
}OfferedPsksWithNoBinders;
```

This extension MUST be present when the handshake is_psk_proposed absent otherwise. If these conditions are not met, CS SHOULD raise a 'invalid_handshake' error.

In addition the ClientHello may also have its KeyShare replaced by PartialKeyShareClientHello as detailed in Section 4.5.

The CS MUST check the handshake is ks_proposed or psk_proposed.

4.2.8. c_server_hello

There is no particular conditions to be met by the handshake.

4.2.9. c_client_finished

The c_client_finished exchange considers two types of handshake messages. Either the handshake message starts with a EncryptedExtensions message when the exchange follows a c_server_hello exchange. This situation corresponds to the case where E c_client_finished follows a c_server_hello. Otherwise the handshake starts with a ServerHello message. This situation corresponds to the case where E c_client_finished follows a c_init_client_hello exchange.

In both cases, Certificate message are not part of the handshake and are instead carried by the Cert structure (see Section 4.7).

4.3. session_id

A TLS handshake and the establishment of a TLS session may involve a single interaction between E and CS or multiple interactions. In the first case, the interactions are called stateless while in the other case, interactions are statefull and the different interactions are managed through a session identified by E and CS with a session_id.

The session_id is a 32 bit identifier that identifies a LURK session between E and the CS. Session_ids are agreed in the first exchange of the session that instantiates the session. The exchange is designate with 's_init_' or 'c_init_' and E (resp. the CS) indicates the value to be used for inbound session_id in the following exchanges. For other exchanges, the session_id is set by the sender to the inbound value provided by the receiving party. Once the session_id has been negotiated by E, all message of the session will have a session_id.

When CS receives an unexpected session_id CS SHOULD return an invalid_session_id error.

Some exchange - indicated with * in Figure 5 - may initiate a session or be part of a stateless interaction between E and CS. More specifically, the s_init_cert_verify exchange instantiates a session only when E expects it to be followed by one or multiple s_new_tickets messages. If the TLS server does not intend to provide session resumption for example, no further s_new_tickets exchange are expected. A similar situation occurs with the c_init_client_finished exchange that may be followed by one or multiple c_post_hand_auth exchanges. If the TLS client does not supports post handshake authentication, no further exchanges are envisioned.

To handle with these case, E indicates a stateless interaction by setting tag.last_exchange to True and MUST omit the session_id. As described in To refuse the establishment of the session, CS sets the

tag.last_message to True and omit the session_id.Section 4.8, CS MUST set the tag.last_message to True and MUST as well and MUST omit the session_id. Reversely, E may also initiates a session, in which case, it sets tag.last_exchange to False and it MUST provide a session_id. Upon receiving the request, CS may establish a session or refuse the establishment of the session and instead consider the stateless mode.

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

Table Figure 5 indicates the presence of the session_id for each exchange.

LURK exchange	session_id
s_init_cert_verify	*
s_new_ticket	y
s_init_early_secret	y
s_hand_and_app_secret	y
c_init_client_finished	*
c_post_hand_auth	y
c_init_client_hello	y
c_server_hello	y
c_client_finished	y
c_register_tickets	y

y indicates the session_id is present
* indicates session_id may be present (depending on the tag.last_exchang

Figure 5: session_id in LURK exchanges

4.4. freshness

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

If CS does not support the freshness, CS SHOULD return an invalid_freshness error.

The freshness structure is defined as follows:

```
enum { sha256(0), sha384(1), sha512(2), ... (255) } Freshness;
```

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

LURK exchange	freshness
s_init_cert_verify	y
s_new_ticket	-
s_init_early_secret	-
s_hand_and_app_secret	y
c_init_client_finished	y
c_post_hand_auth	-
c_init_client_hello	y
c_server_hello	-
c_client_finished	-
c_register_tickets	-

y indicates freshness is present
- indicates freshness is absent

Figure 6: freshness in LURK exchange

When 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 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 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.

4.5. 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:

- *e_generated: Where (EC)DHE keys and shared secret are generated by E and the shared secret is provided to the CS

- *cs_generated: Where the (EC)DH keys and shared secret are generated by CS and the public key is returned to the E.

- *no_secret: where no (EC)DHE is involved, and PSK authentication is performed.

4.5.1. TLS server side

On the TLS server side, the EphemeralRequest and EphemeralResponse structures are defined as follows:

```
enum { no_secret (0), e_generated(1), cs_generated(2) (255)} EphemeralMe

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

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

EphemeralResponse {
    EphemeralMethod method;
    select(method) {
        case cs_generated:
            KeyShareEntry key // KeyShareServerHello.server_share
  }
}
```

Where:

coordinate_length indicates the length in bytes of the shared_secret and 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.

4.5.1.1. e_generated:

When the (EC)DHE keys and (EC)DHE shared secret are generated by the E, E provides the shared secret value to the CS. E indicates a shared secret is provided by sending an EphemeralRequest with a method set to 'e_generated'. The shared secret is transmitted in the SharedSecret structure. The SharedSecret structure is a KeyShareEntry described in [RFC8446] section 4.2.8. where the KeyShareEntry.key_exchange is replaced by SharedSecret.shared_secret.

Upon receiving a EphemeralRequest, CS MUST check group is proposed in the KeyShareClientHello and agreed in the KeyShareServerHello. If not an 'invalid_ephemeral' error SHOULD be raised. CS takes the shared_secret provide as the input to the key schedule. CS MUST NOT return any data and the EphemeralResponse only contains the method. Note that CS is unable to check the (EC)DHE shared secret has been generated with the public keys provided either by the ClientHello or the ServerHello.

4.5.1.2. cs_generated:

When the ECDHE public/private keys are generated by the CS, E sends an EphemeralRequest with a method set to 'cs_generated' to request CS to generate the (EC)DHE private key and return the corresponding key that is returned to the TLS client. The EphemeralRequest structure does not carry any other information than the method. Instead the ServerHello carries the group information in a NameGroup structure. The NameGroup structure is carried by a modified KeyShareEntry of the KeyShareServerHello structure. Note that in such cases CS would receive an incomplete Handshake Context from E with the public part of the ECDHE (KeyShareServerHello.server_share.key_exchange) being of zero length as described by the EmptyKeyShareEntry below:

```
struct {
  NamedGroup group;
  int16 key_exchange=0
} EmptyKeyShareEntry
```

Upon receiving the EphemeralRequest, CS MUST check the group field in the KeyShareServerHello, and get the public value of the TLS client from the KeyShareClientHello. CS performs the same checks as described in [RFC8446] section 4.2.8. 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 E.

4.5.2. no_secret

When the PSK only authentication is chosen, (EC)DHE keys and shared secrets are not needed. To indicate that no ECDHE shared secret is involved E set the EphemeralRequest.methode to 'no_secret'.

Upon receiving the EphemeralRequest, CS MUST check the PSK authentication without (EC)DHE has been agreed. More specifically, CS MUST check the handshake is 'psk_proposed', 'psk_agreed' and is not 'ks_agreed' as detailed in [RFC8446] section 4.2.9.

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

4.5.3. TLS client side

While the TLS server side proceeds to a ephemeral exchange with ephemeral structures in both request and response, in the case of the TLS client, ephemeral only appears in the request or in the response. Typically, in the c_init_client_hello, E requests CS to generate (EC)DHE via a specific structure of the KeyShareEntry carried by the CLientHello. CS on the other hand returns the generated public (EC)DHE keys in a ephemeral structure. In the case of the c_server_hello, E provides the necessary information to CS, but does not receive any response.

4.5.3.1. c_init_client_hello

A TLS client may propose multiple KeyShareEntry to the TLS server in a KeyShareClientHello.client_shares structure. These KeyShareEntry may be generated by E or CS. E indicates CS a (EC)DHE key needs to be generated by replacing the KeyShareEntry by am EmptyKeyShareEntry. E does not provide any EphemeralRequest message as no extra information is needed. Upon receiving a c_init_client_hello, CS processes every KeyShareEntry or EmptyKeyShareEntry of the KeyShareClientHello.client_shares to return a list of EphemeralResponse structure. For each KeyShareEntry or EmptyKeyShareEntry of the client_shares - processed in the exact order of the client_shares. When a KeyShareEntry is encountered, CS generates a corresponding EphemeralResponse with a method set to 'e_generated'. When an EmptyKeyShareEntry is encountered, CS generates the private key and generates a KeyShareEntry that contains the corresponding public key. That KeyShareEntry replaces the EmptyKeyShareEntry and is added to an EmphemeralResponse with a method set to 'cs_generated'. The resulting list of EphemeralResponse is returned to E, so E can complete its CLientHello.

4.5.3.2. c_server_hello

With a c_server_hello exchange, E determine the selected KeyShareEntry by the TLS server KeyShareServerHello.server_share If the (EC)DHE client secret key has been generated by CS, E sends an EphemeralRequest with a method set to 'cs_generated'. If the (EC)DHE client secret key has been generated by E, E provides the share secret via a EphemeralRequest with its method set to 'e_generated'. When no (EC)DHE is being used, E sets its ephemeral to 'no_secret'. A 'no_secret' ephemeral correspond to a psk_agreed handshake only.

Table {table:ephemeral} indicates the exchanges that contain the ephemeral parameter as well as the permitted methods. The CS MUST check compliance with Table {table:ephemeral} and raise an 'invalid_ephemeral' error in case of non compliance.

			method= secret	
LURK exchange	ephemeral	no	e_gen.	cs_gen.
s_init_cert_verify	y	-	y	y
s_new_ticket	-	-	-	-
s_init_early_secret	-	-	-	-
s_hand_and_app_secret	y	y	y	y
c_init_client_finished	y	y	y	-
c_post_hand_auth	y	-		-
c_init_client_hello	y response	-	y	y
c_server_hello	y request	y	y	y
c_client_finished	-	-	-	-
c_register_tickets	-	-		

y indicates possible value for method
- indicates incompatible value for method

Figure 7: Ephemeral field in LURK exchange

4.6. selected_identity

The selected_identity indicates the identity of the PSK used in the key schedule. The selected_identity expresses index of the identities in the in the ClientHello pre_shared_key extension as expressed in [RFC8446] section 4.2.11.

The selected_identity structure is defined as follows:

```
uint16 selected_identity; //RFC8446 section 4.2.11
```

LURK exchange	req
s_init_cert_verify	-
s_new_ticket	-
s_init_early_secret	y
s_hand_and_app_secret	-
c_init_post_hand_auth	-
c_post_hand_auth	-
c_init_client_hello	-
c_server_hello	-
c_client_finished	-
c_register_tickets	-

y indicates the selected_identity is present
- indicates the selected_identity is absent

Figure 8: psk_id in LURK exchange

CS retrieve the PSK identity from the ClientHello and SHOULD send an invalid_psk error if an error occurs. For the TLS server, CS MUST check the selected_identity parameter matches the selected_identity of the ServerHello as described in Section 4.2.

4.7. cert

cert indicates the presence or absence of a Certificate message as well as the necessary input for the generation of such message. cert supports different type of compressed certificates defined by a cert_type.

cert is essentially motivated to enable compression of the Certificate message similarly to CompressedCertificate [RFC8879]. The reason to to use a specific cert parameter is that CompressedCertificate is negotiated between the TLS client and the TLS server - with some implications on teh ClientHello extensions - while here the communication is between E and the CS. All related certificate information are handled by this parameter and Certificate message MUST not be provided via the handshake parameter.

cert supports FingerPrintCertificate to compress the Certificate message when certificates are configured on both E and the CS. This expected to be used for client Certificate exchanged on the TLS client between CS and E or server Certificate exchanged on the TLS server between E and the CS. The compressed format is a FingerPrintCertificate which contains a list of FingerPrintCertificateEntry where each opaque certificate data -

RawPublicKey or X509 content as described in [RFC8446] section 4.4.2 - is instead replaced by the 4 byte finger_print of these certificate data. The finger_print consists in the 4 first bytes of the output hash of the certificate using SHA256 as the hashing function.

cert also supports the CompressedCertificate [RFC8879] or the uncompressed Certificate message format.

Finally, cert also indicates the absence of Certificate message with a special no_certificate type.

CS MUST support the no_certificate, the finger_print an uncompressed cert_type. CS SHOULD raise a 'invalid_cert_type' error when it receives a unsupported cert_type and 'invalid_cert' when any other error occurs.

When (EC)DHE authentication has been agreed (the handshake is not in psk_agreed), CS MUST check the presence of a server Certificate message and reject and raise a 'invalid_certificate' error if cert_type is set to no_certificate. When the handshake is psk_proposed and psk_agreed, CS MUST check the absence of server Certificate and raise an 'invalid_certificate' if cert_type is not set to no_certificate. When the handshake is certificate_request, CS MUST check the presence of a client Certificate message and raise an 'invalid_certificate' if cert_type is not set to no_certificate.

The cert structure is defined as follows:

```
enum { zlib(1),  brotli(2),  zstd(3), no_certificate(128), finger_print(
} CertType;

struct{
    select (certificate_type) {
              case RawPublicKey:
                uint32 finger_print;
              case X509:
                uint32 finger_print;
          };
          Extension extensions<0..2^16-1>;
} FingerPrintCertificateEntry;

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



struct {
    CertType cert_type;
    select (cert_type) {
        case zlib, brotlib, zstd:
          CompressedCertificate certificate; RFC8879 section 4
        case no_certificate:
          // no certificate
        case finger_print
            uint24 uncompressed_length;
            FingerPrintCertificate certificate;
        case uncompressed:
            Certificate certificate; // RFC8446 section 4.4.2
    };
} Cert;
```

certificate structure carrying the compressed form of the Certificate message. The current format supported are CompressedCertificate as detailed in [RFC8879], FingerPrintCertificate, and Certificate certificate as detailed in [RFC8446] section 4.4.2.

certificate_list A sequence (chain) of FingerPrintCertificateEntry structures, each containing a single certificate and set of extensions.

certificate_request_context, extensions, certificate_type are defined in [RFC8446] section 4.4.2.

finger_print

the first 4 bytes of the resulting SHA256 output of the certificate in DER format or the ASN.1_subjectPublicKeyInfo [RFC7250].

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

LURK exchange	cert	certificate type
s_init_cert_verify	y	server
s_new_ticket	y	client*
s_init_early_secret	-	
s_hand_and_app_secret	-	
c_init_client_finished	y	client* / server*
c_post_hand_auth	y	client
c_init_client_hello	-	
c_server_hello	-	
c_client_finished	y	client*/server*
c_register_tickets	-	

y indicates the presence of certificate in the exchange
- indicates the certificate structure is absent
* indicates certificate type MAY be set to no_certificate.

Figure 9: cert in LURK exchange

In the s_new_ticket exchange, if cert_type is set to 'no_certificate', the handshake does not have a CertificateVerify message. Reversely, if cert_type is not set to 'no_certificate' a CertificateVerify message MUST be present in the hanshake.

In the c_init_client_finished, if server_certificate.cert_type is set to 'no_certificate', CS checks the handshake is psk_proposed and psk_agreed. Reversely, if server_certificate.cert_type is not set to 'no_certificate', CS checks the handshake is certificate_agreed. If the client_certificate is set to 'no_certificate', CS checks the handshake is not certificate_request. Reversely, if the client_certificate is not set to 'no_certificate', CS checks the handshake is certificate_request.

If one if these condition is not met, CS raises an 'invalid_certificate' error.

4.8. tag

This field provides extra information. Currently, the tag structure defines tag.last_exchange and tag.cert_request.

E or CS sets the tag.last_exchange to terminate the LURK session. When E is expecting the current exchange to be the last one, it sets the tag.last_exchange to True. CS MUST respond with a tag.last_exchange set to True and E MUST ignore any other values. When E does not expect the current message to be the last one it sets the tag.last_exchange to False. CS responds with a tag.last_exchange set to False to confirm additional exchanges may be performed. On the other hand, CS may also indicates this is the last exchange due to configuration or internal policies and set tag.last_exchange to True. Typically, for the s_new_ticket exchange, it is RECOMMENDED CS is configured with a maximum number of tickets to emit, and to set the tag.last_exchange to True when this limit has been reached. For the c_init_client_finished exchange, the CS MUST set tag.last_exchange to True if the handshake is not post_hand_auth_proposed. For the c_post_hand_auth exchange, it is RECOMMENDED CS is configured with a maximum number of post handshake authentication and set the tag.last_exchange to True once this limit has been reached.

Upon receiving a response with a tag.last_exchange set to True, E MUST reset the session and MUST NOT proceed to any further exchanges.

In this document, we use setting, setting to True to indicate the bit is set to 1. Respectively, we say unsetting, setting to False to indicate the bit is set to 0.

Table Figure 10 indicates the different values carried by the tag as well as the exchange these tags are considered. The bits values MUST be ignored outside their exchange context and bits Bits that are not specified within a given exchange MUST be set to zero by the sender and MUST be ignored by the receiver.

Bit	description
0	last_exchange
1-7	RESERVED

Figure 10: tag description

LURK exchange	last_exchange
s_init_cert_verify	y
s_new_ticket	y
s_init_early_secret	-
s_hand_and_app_secret	y
c_init_client_finished	y
c_post_hand_auth	y
c_init_client_hello	-
c_server_hello	-
c_client_finished	y
c_register_tickets	y

y indicates tag is present
- indicates tag is absent

Figure 11: tag per LURK exchange

c_init_client_finished

4.9. secret

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

The Secret structure is defined as follows:

```
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.

4.10. 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 certificate Section 4.7 and the context is derived from the handshake Section 4.2 and certificate Section 4.7.

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"

The signature structure is defined as follows:

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

5. LURK exchange on the TLS server

This section describes exchanges performed on the TLS server. Unless specified, used structures are described in Section 4

The interaction between E and CS are relatively straight forward as the TLS server is able select how it will be authenticated, that is either using the certificate or the PSK with (EC)DHE or the PSK only. If the TLS server selects to be authenticated via its certificate, E request CS, via a s_init_cert_verify, to generate the signature, as well as handshake, application and exporter secrets to complete the handshake. If the TLS server selects to be authenticated via a PSK E request CS, via a s_init_early_secret, at least the binder key to validate the binders as well as other early secrets to complete the handshake. Once the ticket has been validated, E requests CS the handshake and application secrets and optionally the generation of the (EC)DHE.

Once, the handshake has been completed, E may request the generation of tickets to enable session resumption. This is again a server side decision which may be performed via a s_new_ticket exchange.

The remaining of the section describes CS behavior. Implementation may differ from the description and generates the response otherwise. The response SHOULD however remain coherent with the description provided.

5.1. s_init_cert_verify

s_init_cert_verify initiates a LURK session when the server is authenticated with (EC)DHE. E sends a SInitCertVerifyRequest to CS and is responded a SInitCertVerifyResponse structure unless an error is being raised.

```
struct{
    uint8 tag;
    select (tag.last_exchange){
      case False:
        uint32 session_id;
    }
    FreshnessFunct freshness;
    Ephemeral ephemeral;
    Handshake handshake<0..2^32>; //RFC8446 section 4
    Cert 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.

For other parameters, see the corresponding sections tag (Section 4.8), session_id (Section 4.3), freshness (Section 4.4), ephemeral (Section 4.5), handshake (Section 4.2), certificate (Section 4.7), secret_request (Section 4.1), signature (Section 4.10), secret_list (Section 4.9).

If the exchange is expected to be followed by a s_new_ticket exchange, typically to enable session resumption, E set the tag.last_exchange to False as described in Section 4.8.

When tag.last_exchange is set to False, E indicates in the session_id structure, the session identifier used to further identify the inbound session - see Section 4.3 for more details. If tag.last_exchange is set to True, the session_id structure is ignored by the E.

E sets the freshness function as detailed in Section 4.4. It is RECOMMENDED to use a freshness function that provides similar security as TLS Hash function.

E either generates the (EC)DHE or requests CS to generates, in which case it respectively sets the ephemeral.method to 'e_generated' or 'cs_generated' and proceeds as described in Section 4.5.

E provides the necessary TLS Handshake context in handshake and certificate as respectively detailed in Section 4.2 and Section 4.7.

The necessary secrets E needs to complete the exchange as well as the signature scheme used to generate the signature are indicated by secret_request and sig_algo as described in Section 4.1 and Section 4.10.

Upon receiving the SInitCertVerifyRequest, CS generates the ServerHello.random as detailed in Section 4.4 to implement anti replay protection.

Then, CS generates the (EC)DHE to further initialize the key schedule as described in [RFC8466] section 7.1. As described Section 4.5, the (EC)DHE is either directly provided by E ('e_generated') or computed by CS ('cs_generated'). When (EC)DHE are generated by the CS, necessary information are provided by the partial key_share extension. CS completes the key share extension and returns that necessary information to E in the ephemeral response.

Once (EC)DHE is computed, CS generates the handshake secrets (h_c, h_s) as described in [RFC8446] section 7.1, compute the Certificate message from the certificate and complete the TLS handshake in order to have the necessary TLS Handshake context to generate the signature.

CS then generates the signature, provides the resulting value in the signature field of the response and complete the CeriticateVerify message that inserted into the TLS handshake context of the CS, so the Finished message can be generated. This provides the necessary TLS Handshake context for the generation of the application secrets (a_c, a_s)

Once generated, the requested secrets are returned in secret_list as detailed in Section 4.1 and Section 4.9.

5.2. 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 CS to E as the possession of the resumption_master_secret (r) is sufficient to proceed to session resumption. It is important to realize that performing session resumption outside CS may have some security implications - especially when CS provides a more secure environment than the E. CS MAY responds with a resumption_master_secret based on its policies.

The LURK client MAY perform multiple s_new_ticket exchanges.

The SNewTicketRequest and SNewTicketResponse are described below:

```
struct {
    uint8 tag
    uint32 session_id
    Handshake handshake<0..2^32> //RFC8446 section 4.
    Cert 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. CS MAY responds with less tickets when the value is too high.

For other parameters, see the corresponding sections tag (Section 4.8), session_id (Section 4.3), handshake (Section 4.2), certificate (Section 4.7), secret_request (Section 4.1), secret_list (Section 4.9).

When E is requesting CS to generate tickets, as described in Section 4.3 E sets the session_id with the value received in the SInitCertVerifyResponse or the SInitEarlySecretResponse of the previous exchange.

E MUST ensure CS has sufficient TLS Handshake context, that is the client Finished and optionally the client Certificate and

CertificateVerify messages have been provided. When these messages have been already provided, E SHOULD NOT provide them again and CS MAY upon configuration raise an invalid_handshake error. In such case, E may resend its request with the appropriated empty handshake and certificate.

Note that client Certificate are carried via certificate, which enables to compress the Certificate payload.

E may set the secret_request to 'r'.

Upon receiving a SNewTicketRequest CS check the tag.last_exchange and define if further messages are expected or not by setting the tag.last_exchange in its response. The session_id is set to the value provided in the SInitCertVerifyRequest or the SInitEarlySecretRequest of the previous exchange.

If CS does not have sufficient handshake context and invalid_handshake error is raised as described in Section 4.2. If the ticket_nbr exceeds the maximum number of ticket authorized by the CS, an authorized lower number of tickets is returned and if not further tickets can be requested, the tag.last_exchange MUST be set to True to close the session. If a lower number of ticket is returned while tag.last_exchange is set to False, E interpret it as a maximum number of ticket per transaction, and may initiate further s_new_tickets exchanges.

5.3. s_init_early_secret

s_init_early_secret initiates a LURK session when the server is authenticated by the PSK or PSK-ECDHE methods.

The SInitEarlySecretRequest and SInitEarlySecretResponse are define dbelo9w:

```
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
```

selected_identity indicates the selected PSK as detailed in Section 4.6.

For other parameters, see the corresponding sections session_id (Section 4.3), freshness (Section 4.4), handshake (Section 4.2), secret_request (Section 4.1), secret_list (Section 4.9).

When E agrees to perform a PSK base server authentication, it sends a SInitEarlySecretRequest to the CS. The session_id are generated as described in the s_init_cert_verify exchange (see Section 5.1. The binder_key MUST be requested, since it is used to validate the PSK. If the TLS client has indicated support for early application data via the early_data extension and if the TLS server enable early_data, E requests the client_early_traffic_secret (e_c). E MAY also request the early_exporter_master_secret (e_x).

CS MUST check the handshake corresponds to a PSK authentication, that is the handshake is, for the partial ClientHello, the handshake is psk_proposed - see Section 4.2 for more details. CS selects the PSK indicated by the selected_identity and initiates the key scheduler before generating the requested secrets. The key schedule is instantiated with the PSK and TLS hash function provided by the ticket. As detailed in Section 4.1, CS MUST generate the binder_key and check the binders. If e_c has been requested, CS MUST check the presence of an early_data extension in the ClientHello before generating e_c (see Section 4.2). The generation of both e_c and e_x is subject CS policies.

5.4. s_hand_and_app_secret

The s_hand_and_app_secret follows the s_init_early_secret exchange and enable the generation of h_c, h_s, a_c, a_s and x. When the (EC)DHE private key is generated by the CS, this exchange also provides the corresponding public key necessary to complete the ServerHello.

The SHandAndAppSecretRequest and SHandAndAppSecretResponse structures are described below:

```
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
```

Parameters are defined in their corresponding sections tag (Section 4.8), session_id (Section 4.3), ephemeral (Section 4.5), handshake (Section 4.2), secret_request (Section 4.1), secret_list (Section 4.9).

To send a SHandAndAppSecretRequest, E sets the tag.last_exchange as described in Section 5.1. As described in Section 4.3 E sets the session_id with the value received in the SInitEarlySecretResponse. If E does decides the TLS server is authenticated with the PSK mode without (EC)DHE, it sets ephemeral.method to 'no_secret'. If the TLS server is authenticated with PSK and (EC)DHE, then the ephemeral.method can be set to 'e_generated' or 'cs_generated' depending whether E or CS generates the (EC)DHE private key.

Upon receiving the SHandAndAppSecretRequest sets the tag.last_exchange as described in Section 5.1. The session_id is set to the value provided in the SInitEarlySecretRequest. ephemeral is generated as described in Section 5.1 when ephemeral method is set to 'e_generated' or 'cs_generated' and a method set to 'no_secret' does not trigger any action. CS MUST check the handshake is coherent both to PSK or PSK with (EC)DHE as well as the ServerHello is coherent with the choices indicated in the previous SHandAndAppSecretRequest. More specifically, the handshake must be psk_agreed for an PSK authentication without (EC)DHE. When PSK authentication with (EC)DHE has been selected, the handshake MUST be psk_agreed ks_proposed and ks_agreed. Then CS MUST also ensure the selected_identity indicated in the ServerHello corresponds to the one provided in the SInitEarlySecretRequest and that the cipher_suite of the ServerHello has the same hash function as the one provided by the ticket - see Section 4.2.

CS generates the ServerHello.random as detailed in Section 4.4 to implement anti replay protection. Withe the ServerHello message set, the handshake secrets are generated with the key schedule initiated

during the s_init_early_secret. CS then generates the Finished message before generating the application and exporter secrets.

6. LURK exchange on the TLS client

Figure Figure 12 summarizes the different possible LURK session as well as the different messages that are involved in the session.

Similarly to the TLS server, the credentials involved in the TLS client authentication are PK, (EC)DHE and PSK. This document assumes PK, when involved, is always protected by the CS and considers the three following scenarios:

- CS only protects PK in which case CS is limited to generating the signature of the CertificateVerify of the TLS client. Such signature may be generated during the TLS handshake or during a client post handshake authentication.

- CS protects the (EC)DHE key or the PSK in addition to the PK. In particular, resumed session are provided a similar level of security by CS.

The protection of the (EC)DHE or the PSK requires some interaction between E and the CS to build the ClientHello message (c_init_client_hello), as well as once the server Finished is received to build the optional CertificateVerify and client Finished message (c_client_finished). Once the TLS handshake is finished, E and CS may interact to register tickets received by the TLS server (c_register_ticket) or to proceed to a client post handshake authentication when requested by the TLS server and when permitted by the TLS client (c_post_hand_auth).

On the other hand, when only PK is protected, the generation of signature by the CS may occur for the client authentication (during the TLS handshake) or after, during a client post handshake authentication. In both cases, E interacts with CS after the server Finished is received. When the TLS server request the client authentication, E requests CS to generate the necessary signature for the client CertificateVerify message. When the TLS server does not request the client authentication and that TLS client has previously indicated the support for post handshake authentication, E provides to CS the necessary context to later perform the post handshake authentication. More specifically, a interaction between E and the CS during a post handshake authentication only would either require E to provide the full handshake context which would cause too much penalty to implementation that only keep the transcript of the handshake. The alternative of providing the necessary transcript to generate the signature would prevent the CS to control the input that is actually signed as well as to provide anti-replay protection

as detailed in Section 4.4. When the TLS client has indicated the support for post handshake authentication, the only permitted exchange is c_post_hand_auth.

(a) (method is 'e_generated' or 'no_secret') and chosen PSK not in CS

Figure 12: LURK client State Diagram

6.1. c_init_client_finished

E initiates a c_init_client_finished exchange during a TLS handshake when it receives a server Finished message an meets the following conditions:

1. The TLS handshake has not required any interaction between E and CS.

2. The TLS server requests a client authentication or the TLS client has enabled client post handshake authentication.

The CInitClientFinishedRequest and CInitClientFinishedResponse are detailed below:

```
struct{
    uint8 tag;
    select (tag.last_exchange){
      case False:
        uint32 session_id;
    }
    Handshake handshake<0..2^32>; //RFC8446 section 4
    Cert server_certificate;
    Cert client_certificate
    FreshnessFunct freshness;
    Ephemeral ephemeral; ## e_generated
    opaque psk<0..2^16>;
}CInitClientFinishedRequest

struct{
    uint8 tag;
    select (tag.last_exchange){
      case False:
        uint32 session_id;
    }
    Signature signature;
}CInitClientFinishedResponse
```

psk when provided, the PSK to be used for the schedule as described in [RFC8446] section 7.1.

If the TLS handshake is not post_hand_auth_proposed, E set tag.last_exchange to True, otherwise, tag.last_exchange SHOULD be set to False.

The Handshake consists in every messages up to the server Finished message to which the server Certificate message, if present has been omitted and is instead carried in the server_certificate. If handshake is psk_agreed, server_certificate.cert_type is set to 'no_certificate', otherwise another cert_type MUST be used. client_certificate.cert_type is set to 'no_certificate' in the absence of CertificateRequest message and other type MUST be used otherwise.

If the TLS server is authenticated with a certificate or using PSK-ECDHE, the (EC)DHE is generated by E and ephemeral.method is set to 'e_generated'. If the TLS server is authenticated with PSK, the ephemeral.method is set to 'no_secret'.

This exchange occurs when E is performing the key schedule. E SHOULD provide the PSK value unless E knows CS is aware of the PSK or

tag.last_message is set to True. In fact the key schedule is only necessary to enable post handshake authentication. If neither E nor CS are aware of PSK, the TLS client MUST NOT propose post handshake authentication.

Upon receiving a CInitClientFinishedRequest, CS place the server_certificate and client_certificate with a cert_type different from 'no_certificate' in the handshake as described in Section 4.7. CS generates the ClientHello.random with the freshness function as described in Section 4.4. If the handshake is certificate_request, CS generates the signature.

CS sets tag.last_exchange and session_id as respectively detailed in Section 4.8 and Section 4.3.

If tag.last_exchange is False, CS generates the client CertificateVerify and client Finished message, so further post handshake authentication can be handled. The generation of the Finished message requires the h_c to be computed key schedule as described in [RFC8445] section 7.1. The PSK used in the key schedule is determine as follows by CS. If the CS has any out of band knowledge of the PSK to be used, CS SHOULD use that value and ignore the value provided by E. If CS does not have such knowledge, CS takes the value provided E. A zero length PSK is considered as the default value as described in [RFC8446] section 7.1.

6.2. c_post_hand_auth

If the TLS client has indicated support for post handshake client authentication, the TLS client MAY receive a CertificateRequest from the TLS server.

To proceed to the requested authentication, E proceeds to a c_post_hand_auth exchange with the CS.

The CPostHandAuthRequest and CPostHandAuthResponse are detailed below:

```
struct{
  Tag tag
  uint32 session_id
  Handshake handshake<0..2^32> // CertificateRequest
  Cert client_certificate;
}CPostHandAuthRequest


struct{
  Tag tag
  uint32 session_id
  Signature signature
}CPostHandAuthResponse
```

E indicates with tag if it is expecting CS to proceed to additional post handshake client authentication - see Section 4.8. session_id is set to the negotiated session_id provided by the CS for its inbound traffic - see Section 4.3. handshake contains the CertificateRequest received by the TLS server, and sig_algo the signature scheme selected by E and N which indicates the client_application_traffic_secret_N used to be used.

Upon receiving a CPostHandAuthRequest, CS checks the handshake is post_hand_auth_proposed as described in Section 4.2 and SHOULD return an 'invalid_handshake' error otherwise. If the CS permits a limited number of post handshake client authentication, and that limit has been reached, it SHOULD raise an 'invalid_type' error. CS MUST check client_certificate.cert_type is not set to 'no_certificate', the provided certificate is known to CS. The certificate is decompressed and added to the handshake context as described in Section 4.7. Note also that for the client, the Certificate includes the certificate_request_context provided by the CertificateRequest message as described in [RFC8446] section . The signature is computed as detailed in [RFC8446] section 4.4. CS MAY have a limit of permitted number of post handshake, and set tag.last_exchange to True when that limit is reached.

6.3. c_init_client_hello

The c_init_client_hello occurs when E requires CS to provide some information to complete the ClientHello. This occurs when the CS is generating the (EC)DHE private key as well as when CS protects the PSK and needs the binder key to generate the binders.

The CInitClientHelloRequest and CInitClientHelloResponse are detailed below:

```
enum { sha256(0), sha384(1), sha512(2), ... (255) } TLSHash;

enum { external(0), resumption(1) } PSKType;

struct{
  uint16 identity_index;
  TLSHash tls_hash;
  PSKType psk_type<0..2^16-1>;
  opaque psk_bytes<0..2^16>;
}PSKIdentityMetadata

struct{
  uint32 session_id;
  Handshake handshake<0..2^32>; //RFC8446 section 4
  Freshness freshness;
  PSKMetadata psk_metadata_list<0..2^16>
  uint16 secret_request;
}CInitClientHelloRequest

struct{
  uint32 session_id
  Ephemeral ephemeral_list<0..2^16-1>
  Secret binder_key_list<0..2^16-1>;
  Secret secret_list<0..2^16-1>;
}CInitClientHelloResponse
```

E generates the ClientHello.random as described in If E requires CS to generate (EC)DHE private keys, E replaces the KeyShareEntry structure by the EmptyKeyShareEntry structure as described in Section 4.4.Section 4.5.

If PSK is proposed, E replaces the OfferedPsk structure by the OfferedPsksWithNoBinders defined Section 4.2. The structure contains the PSK identities the TLS client intend to propose the TLS server. These identities may combine a set of PSK protected by CS or not protected by CS. For each PSK not protected by CS, E generates a PSKIdentityMetadata structure. The PSKIdentityMetadata provides the necessary element to instantiates a key schedule. In addition, it contains a identity_index whic is the index of the PSKIdentity in the OfferedPsksWithNoBinders.identities This structure is then added to the psk_metadata_list, and structures MUST respect a strict identity_index increasing order.

Upon receiving a CInitClientHello, CS proceeds to the session_id negotiation and provides its inbound session_id as detailed in Section 4.3. CS takes the handshake message, update the ClientHello.random as described in Section 4.4. The CS extract the KeyShareClientHello.client_shares. When an EmptyKeyShareClientHello structure is encountered, CS generates a corresponding private key

and generates the corresponding KeyShareEntry. KeyShareEntry is returned via a EphemeralResponse with method set to 'cs_generated'. When a KeyShareEntry is found, an EphemeralResponse with method set to 'e_generated' is generated. As generated EphemeralResponse are returned as described in Section 4.5 in the ephemeral_list. If CS is unable to generate an ephemeral of if the number of ephemeral values to generate is too high, a 'invalid_ephemeral' error SHOULD be raised.

If the handshake is 'psk_proposed', CS computes the binder key associated to each PSK present in the OfferedPsks[identities]. The computation of the binder_key only requires the knowledge of the PSK as described in [RFC8446] section 7.1. The computed binder_keys are returned in the binder_key_list following the exact same order as the OfferedPsks[identities]. Once all binder_keys have been generated, CS generates the binders, which requires a partial ClientHello and a binder_key. The binders are integrated to the resulting ClientHello.

With the ClientHello, early secrets are generated and returned as described in Section 4.1.

Note that the binder key MUST be part of the binder_key_list and MUST Be omitted in the secret_list.

6.4. c_server_hello

This exchange follows the c_init_client_hello exchange and the main purpose is the generation of the handshake secrets so E can decrypt the encrypted message sent by the TLS server.

The motivation for handling the decryption by E instead of the CS is to enable E to perform some checks and transformations over the encrypted message. The down side is that it introduces an additional exchange.

The CServerHelloRequest and CServerHelloResponse structures are described below:

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

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

Upon receiving the ServerHello, E checks if the TLS server authentication is only based on PSK without (EC)DHE. This is performed by checking the TLS handshake is psk_proposed, psk_agreed and not ks_agreed. In that case, E sets the EphemeralRequest.method to 'no_secret'. Otherwise, E evaluates the KeyShareEntry selected by the TLS server, that is the KeyShareServerHello.server_share. If the corresponding (EC)DHE private key has been generated by E, E generates the shared key and provide it via an EphemeralRequest structure with a method set to 'e_generated'. If the corresponding (EC)DHE private key has been generated by CS, E sends an EphemeralRequest with a method set to 'cs_generated'.

Note that the handling enables the use of a combination of PSKs hosted by CS and PSK hosted by E. The implication of a PSK hosted by the CS requires a c_init_client_hello exchange. However, the TLS server may select a PSK hosted by E. When in addition the authentication is using PSK without (EC)DHE or when the (EC)DHE has been generated by E, E is able to generate the handshake, the application and resumption secrets. The only remaining interaction between E and CS in the client authentication and these interactions happens during the c_client_finished. In this case E MUST directly perform a c_client_finished exchange.

Upon receiving an CServerHelloRequest, CS checks the method is aligned with the TLS handshake. When the TLS server is authenticated with a certificate, CS initiates the key schedule, otherwise CS continue with the key schedule initialized during the c_init_client_hello and generates the handshake secrets.

Upon receiving a CServerHelloResponse, E decrypt the encrypted messages and proceed to the generation of the application secrets via a c_client_finished exchange.

6.5. c_client_finished

The computation of the application and resumption secret are performed as described in Section 6.1. The only difference is that the key scheduler has already been initialized when it follows a c_server_hello exchange.

The CClientFinishedRequest and CClientFinishedResponse are detailed below:

```
struct{
    uint8 tag;
    uint32 session_id;
    Handshake handshake<0..2^32>; //RFC8446 section 4
    Cert server_certificate;
    Cert client_certificate
    uint16 secret_request;
}CClientFinishedRequest

struct{
    uint8 tag;
    uint32 session_id;
    Signature signature;
    Secret secret_list<0..2^16-1>;
}CClientFinishedResponse
```

E performs a c_client_finished exchange either after a c_server_hello exchange or after a c_init_client_hello exchange. As described in Section 6.4 E performs a c_server_hello to retrieve the handshake secrets necessary to decrypt the encrypted messages of the handshake.

When E has access of the optional PSK and optional (EC)DHE values - or these values are not needed, than E SHOULD compute the secret without any interactions with CS and interaction with CS are restricted to the generation of the signature. In any other cases, E is expected to request a_c, a_s and potentially x and r secrets. Assuming PK is protected by the CS, E performs a c_client_finished either when the TLS server requests the TLS client to authenticate (with a CertificateRequest) or if the TLS client has enabled post handshake client authentication. In this case, CS MUST NOT regenerate resumption secret and MUST sets its last_exchange to True unless post authentication is enabled. In this case handshake starts with a ServerHello message.

Upon receiving a CClientFinishedRequest, CS generates the handshake context as in a c_init_client_finished exchange (see Section 6.1) and generates the signature. More specifically, it generates and insert the server and client Certificate from the server_certificate and client_certificate structures.

6.6. 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. As teh NewSessionTicket's identities may collide when being provided by multiple TLS servers, CS provides identities it manages to prevent

such collisions (CPskID). One such CPskID is assigned to each ticket and is later used to designate that ticket (see Section 6.3). When too many tickets are provided, CS SHOULD raise a too_many_identities error.

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

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

7. 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.

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. 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 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.

8. IANA Considerations

9. Acknowledgments

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport

Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <https://www.rfc-editor.org/info/rfc7250>.

[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>.

[RFC8445] Keranen, A., Holmberg, C., and J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal", RFC 8445, DOI 10.17487/RFC8445, July 2018, <https://www.rfc-editor.org/info/rfc8445>.

[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>.

[RFC8448] Thomson, M., "Example Handshake Traces for TLS 1.3", RFC 8448, DOI 10.17487/RFC8448, January 2019, <https://www.rfc-editor.org/info/rfc8448>.

[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>.

[RFC8879] Ghedini, A. and V. Vasiliev, "TLS Certificate Compression", RFC 8879, DOI 10.17487/RFC8879, December 2020, <https://www.rfc-editor.org/info/rfc8879>.

10.2. Informative References

[I-D.mglt-lurk-lurk] Migault, D., "LURK Protocol version 1", Work in Progress, Internet-Draft, draft-mglt-lurk-lurk-01, 26 July 2021, <https://www.ietf.org/archive/id/draft-mglt-lurk-lurk-01.txt>.

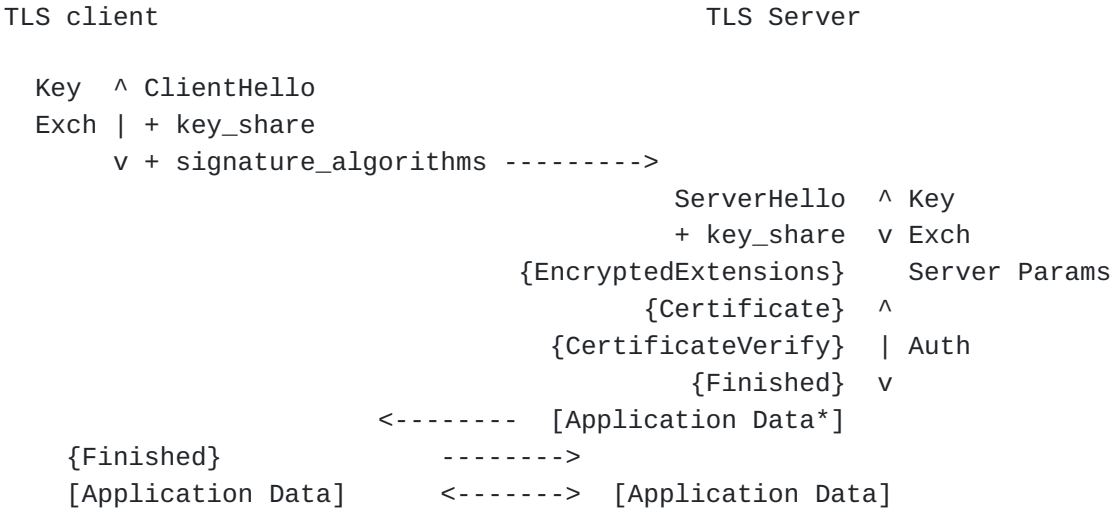
[I-D.mglt-lurk-tls12] Migault, D. and I. Boureanu, "LURK Extension version 1 for (D)TLS 1.2 Authentication", Work in Progress, Internet-Draft, draft-mglt-lurk-tls12-05, 26 July 2021, <https://www.ietf.org/archive/id/draft-mglt-lurk-tls12-05.txt>.

Appendix A. Annex

A.1. TLS server ECDHE (no session resumption)

This section illustrates the most common exchange of a TLS client authenticates a TLS server with it certificate (ECDHE) without session resumption.

The TLS handshake is depicted below from {!RFC8446}}. For clarity as ECDHE authentication is performed, PSK related extensions (psk_key_exchange_modes, pre_shared_key) have been omitted. In addition, as the TLS client is not authenticated, CertificateRequest sent by the TLS server as well as Certificate and CertificateVerify sent by the TLS client have been removed.



The TLS server interacts with CS with a s_init_cert_verify exchange in order to respond to the ClientHello.

Since there is no session resumption, the request 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. The Certificate 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 CS as described in {sec:ex:srv: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}

A.1.1. ecdhe generated on CS

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

In this scenario, CS is the only entity that knows the private ECDHE key and the shared secret, and only CS is able to compute the secrets. CS indicates the exchange is final by setting the tag to last_exchange, returns the x25519 public key that will be included in the ServerHello key_share extension, the signature sig that will be returned in the CertificateVerify message as well as the secrets that will be used to derive the appropriated keys.

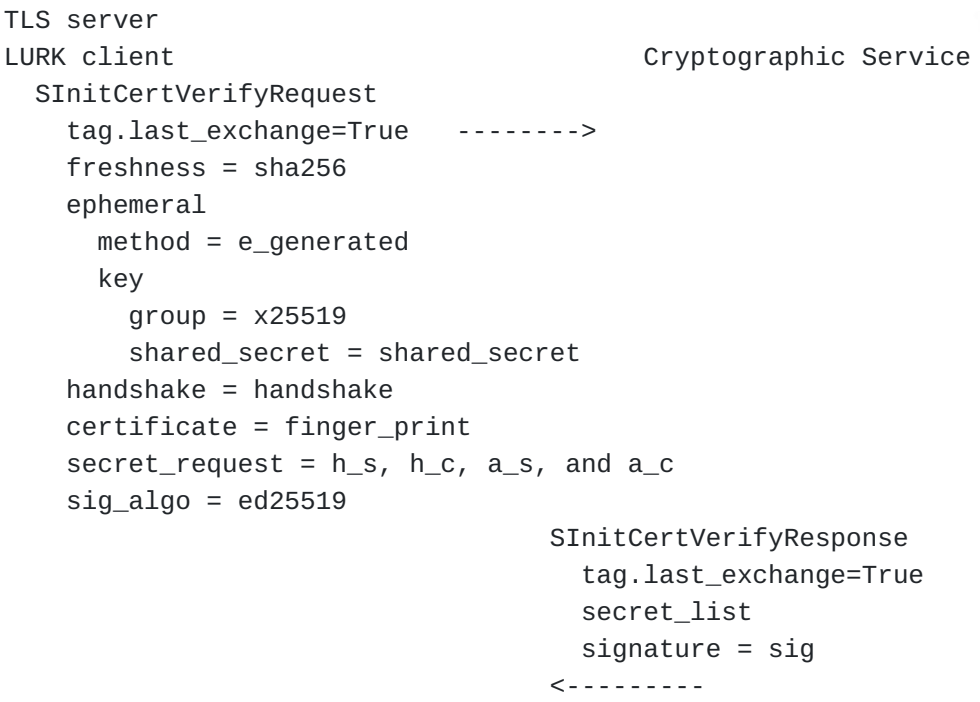
```
TLS server
LURK client                                  Cryptographic Service
  SInitCertVerifyRequest
    tag=last_exchange          -------->
    freshness = sha256
    ephemeral
      method = cs_generated
    handshake = handshake (x25519)
    certificate = finger_print
    secret_request = h_s, h_c, a_s, and a_c
    sig_algo = ed25519
                                    SInitCertVerifyResponse
                                      tag=last_exchange
                                      ephemeral
                                        key
                                          group = x25519,
                                          key_exchange = public_key
                                      secret_list
                                      signature = sig
                               <---------
```

A.1.2. ecdhe generated by the TLS 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 CS as only the shared secret is necessary to generated the signature. This is indicated by the method set to e_generated. No (EC)DHE values are returned by 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 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 the resumption secret, these sessions could be authenticated by the TLS server without any implication from the CS.

In this scenario, CS is the only entity that knows the private ECDHE key. Only CS is able to generate the signature. Both CS and the TLS server are able to compute all secrets. CS indicates the exchange is final by setting the tag to last_exchange, returns the signature sig that will be returned in the CertificateVerify message as well as - when requested - the secrets that will be used to derive the appropriated keys.

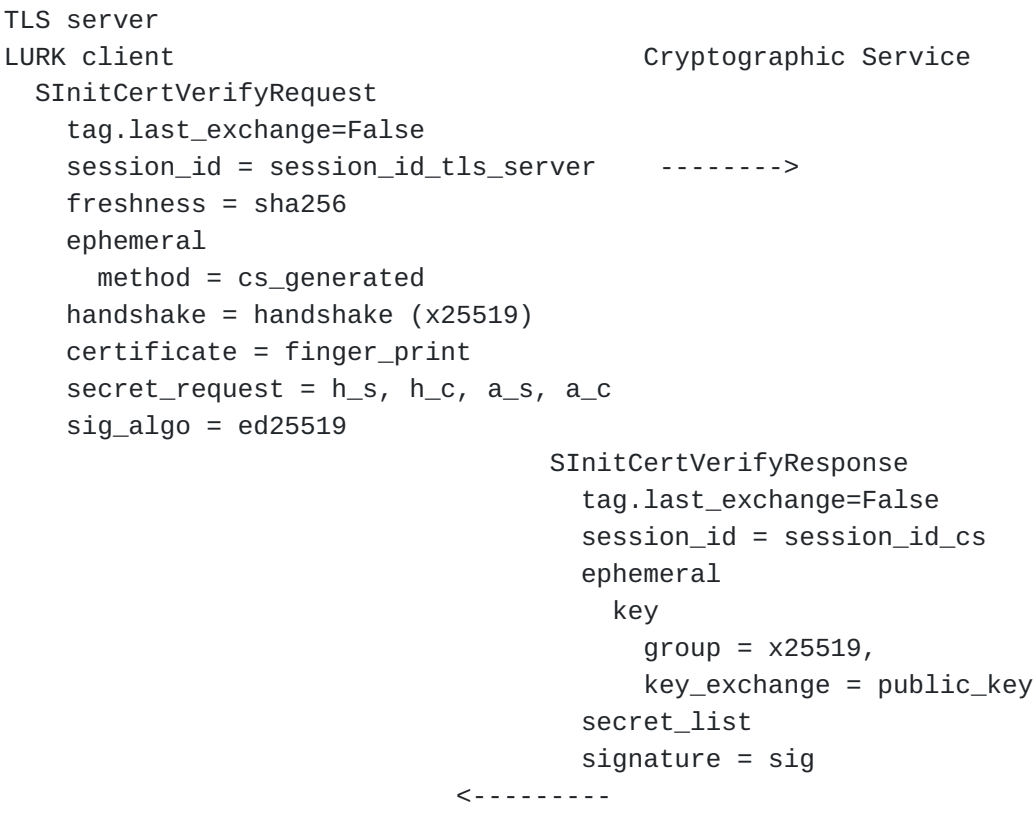


A.2. TLS server ECDHE (with session resumption)

When the TLS client is enabling session resumption, the TLS server is expected to generate some tickets that will be later used for later sessions. The generation of the tickets is based on the resumption_master_secret. To ensure protection of the authentication credential used for the session resumption, CS necessarily must have generated the (EC)DHE keys and must not have provided the resumption_master_secret. In either other cases, the TLS client is able to compute the resumption_master_secret and so session resumption is out of control of the CS. As a result, CS sort of achieves a delegation to the TLS server.

In the remaining of this section, we consider the session resumption is performed by the CS.

ECDHE authentication is performed with CS generating the private part of the (EC)DHE as described in {sec:ex:srv:cs_generated}. However, additional s_new_ticket exchanges are needed so the TLS server provides sufficient material to generate the tickets by CS and retrieves the generated tickets by the CS. As result, the main difference with the scenario described in {sec:ex:srv:cs_generated} is that tag carries a session_id to identify the session between the TLS server and the CS.

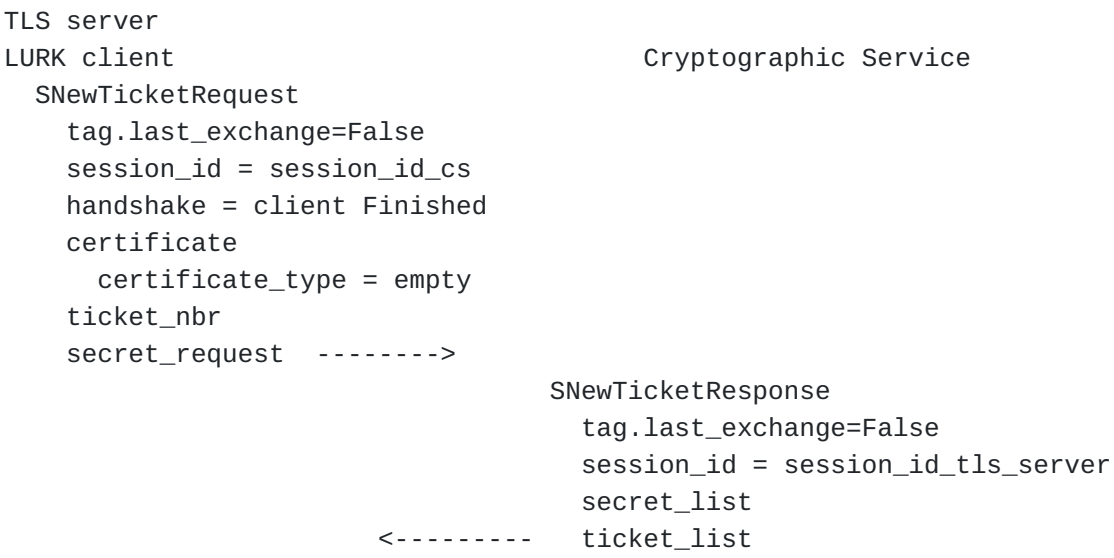


To enable session resumption, the TLS server needs to send NewSessionTickets to the TLS client. This exchange is taken from [RFC8446] and represented below: ~~~ TLS client TLS Server <----- [NewSessionTicket] ~~~

The TLS server requests NewSessionTicket to CS by sending a SNewTicketRequest. The tag.last_exchange set to False indicates to CS the TLS server is willing to request NewSessionTickets multiple times. The session_id is set to the value provided previously by the CS. This session_id will be used to associate the SNewTicketRequest to the specific context of teh TLS handshake. handshake is the remaining handshake necessary to generate the secrets. In some cases, when the TLS client is authenticated, the TLS handshake contains a Certificate message that is carried in the certificate structure as opposed as to the handshake structure. In our current case, the TLS client is not authenticated, so the certificate_type is set to 'empty'. ticket_nbr is an indication of the number of

requested NewSessionTicket, and secret_list indicates the requested secrets. In our case the resumption_master_secret (r) will remain in CS and will be anyway ignored by the CS, so the secret_request has its r bit unset.

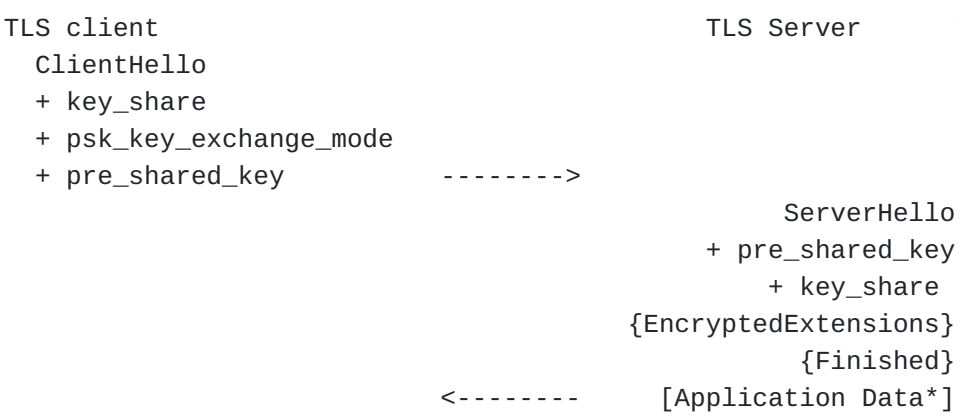
As depicted below, CS provides a list of tickets that could be later used in order to authenticate the TLS server using PSK or PSK-ECDHE authentication as describe din {sec:ex:srv:server-psk}.



A.3. TLS server PSK / PSK-ECDHE

PSK/PSK-ECDHE authentication is the method used for session resumption but can also be used outside the scope of session resumption. In both cases, the PSK is hosted by the CS.

The PSK authentication can be illustrated by the exchange 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 indicates that PSK-ECDHE has been selected.

While the TLS handshake is performed in one round trip, the TLS server and CS have 2 LURK exchanges. These exchanges are consecutive and performed in the scope of a LURK session. A first exchange (s_init_early_secret) validates the ClientHello receives by the TLS server and existence of the selected PSK (by the TLS server) is actually hosted by the CS. Once the s_init_early_secret exchange succeeds, the TLS server starts building the ServerHello and requests the necessary parameters derived by CS to complete the ServerHello with a second exchange (s_init_hand_and_apps).

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 SInitEarlySecretRequest structure provides the session_id that will be used later by the TLS server to identify the session with future inbound responses from CS (session_id_server). The freshness function (sha256) is used to implement PFS together with the ClientHello.random. selected_identity indicates the PSK chosen by the TLS server among those proposed by the TLS client in its ClientHello. The secrets requested by the TLS server are indicated in secret_request. This example shows only the binder_key, but other early secrets may be requests as well.

CS responds with a SInitEarlySecretResponse that contains the session_id_cs used later to identify the incoming packets associated to the LURK session and the binder_key.

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 SHandAndAppSecretRequest structure carries a tag with its last_exchange set to False to indicate the willingness of the TLS server to keep the session open and proceed to further LURK exchanges. In our case, this could mean the TLS server expects to request additional tickets. The session_id is set to session_id_cs, the value provided by the CS. ephemeral is in our case set the method to cs_generated as described in Appendix A.1. The method (x25519) to generate the (EC)DHE is indicated in the handshake. The necessary handshake to derive the handshake and application secrets, as well the requested secrets are indicated in the secret_request structure.

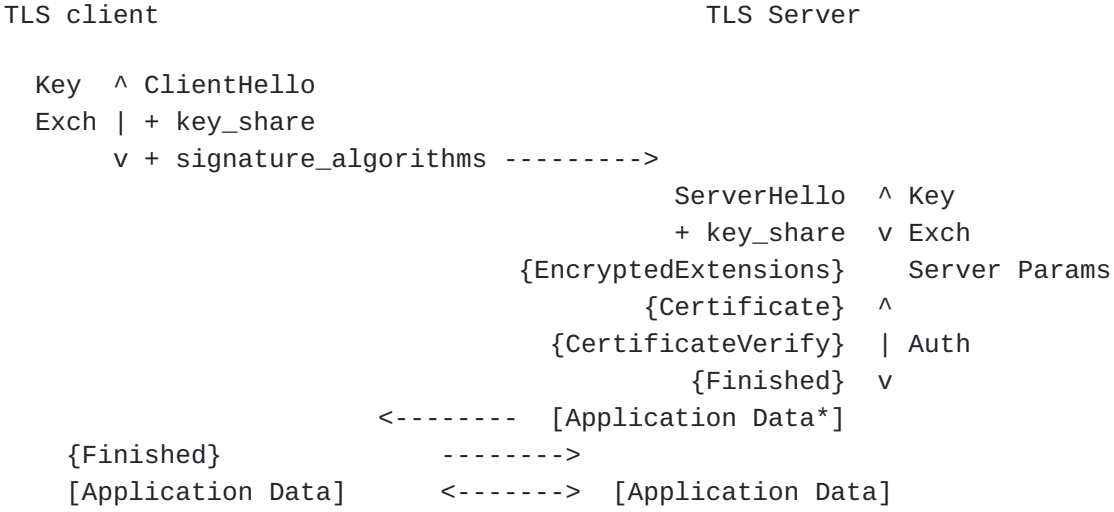
CS sets its tag.last_exchange to True to indicate the session will be closed after this exchange. This also means that no ticket will be provided by the CS. CS returns the (EC)DHE public key as well as requested secrets in a SHandAndAppResponse structure similarly to what is being described in {sec:ex:srv:ecdhe}.

```
TLS server
LURK client                                  Cryptographic Service
  SHandshakeAndAppRequest
    tag.last_exchange = False
    session_id = session_id_cs
    ephemeral
      method = cs_generated
    handshake = ServerHello(x25519) ... EncryptedExtensions
    secret_request = h_c, h_s, a_c, a_s ------->
                                    SHandAndAppResponse
                                      tag.last_exchange = True
                                      session_id = session_id_tls_server
                                      ephemeral
                                        key
                                          group = x25519,
                                          key_exchange = public_key
                           <--------- secret_list
```

A.4. TLS client unauthenticated ECDHE

This section details the case where a TLS client establishes a TLS session authenticating the TLS server using ECDHE. The TLS client interacts with CS in order to generate the (EC)DHE private part. While this section does not illustrates session resumption, the TLS client is configured to proceed to session resumption which will be described with further details in Appendix A.5.

The TLS handshake described in [RFC8446] is depicted below. In this example, the TLS client proposes a key_share extension to agree on a (EC)DHE shared secret, but does not propose any PSK.

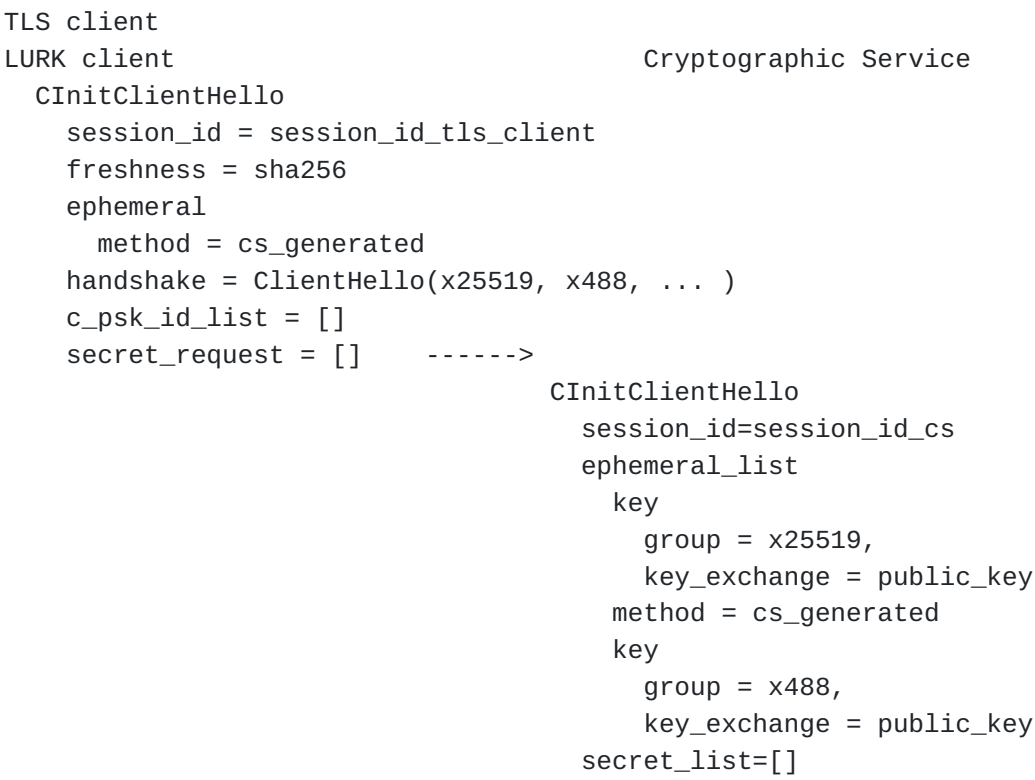


If the TLS client generates the (EC)DHE private key, no interaction with CS is needed as it will have the default PSK value as well as the (EC)DHE shared secrets necessary to proceed to the key schedule described in section 7.1 of [RFC8446].

In this example, the TLS client requests CS via a c_init_client_hello to generate the (EC)DHE private key and provide back the public part that will be placed into the key_share extension before being sent to the TLS server.

Like in any init methods, the TLS client indicates with session_id_tls_client the idnetifier of the session that is being assigned by the TLS client for future inbound LURK message responses sent by the CS. Similarly, CS advertises its session_id_cs. freshness is set to sha256, and the ClientHello.random is generated as described in Section 4.4. handshake contains the ClientHello message to which the key_exchange of the KeyShareentries has been stripped off without changing the other fields. As PSK are not involved, no early secrets are involved and c_psk_list and secret_request are empty.

CS provides the KeyShareEntries. The TLS client is able to build the ClientHello to the TLS server with ClientHello.random and by placing the KeyShareEntries.



Upon receiving the response from the TLS server, responds with a ServerHello followed by additional encrypted messages.

The TLS client needs the handshake secrets to decrypt these encrypted messages and send back the client Finished message. In addition, the TLS client requests the application secrets to encrypt and decrypt the TLS session. The secrets are requested via a c_hand_and_app_secret.

We assume the TLS client supports session resumption so, the tag.last_exchange is unset. The session_id takes the value advertises by each party during the previous c_init_client_hello exchange. Since CS already has the (EC)DHE private keys, it will be able to derive the (EC)DHE shared secret and no information needs to be provided by the TLS client. As a result, method is set to no_secret. The handshake is composed of the messages sent by the TLS server. As the TLS client does not have yet the messages are not decrypted, and are provided encrypted. The requested secrets are the handshake and application secrets.

CS generates the handshake secrets and the associated key to decrypt the encrypted messages. As no CertificateRequest has been found, CS does not compute the signature that would authenticate the TLS client. In this section, we assume CS is ready to accept further exchanges, and in our case the c_register_tickets exchange to enable session resumption. Since session resumption is enabled, CS computes the Finished message to generate the resumption_master_secret.

CS returns the response by unsetting the tag.last_exchange and cert_request. The ephemeral is an empty list and secret_request returns the requested secrets.

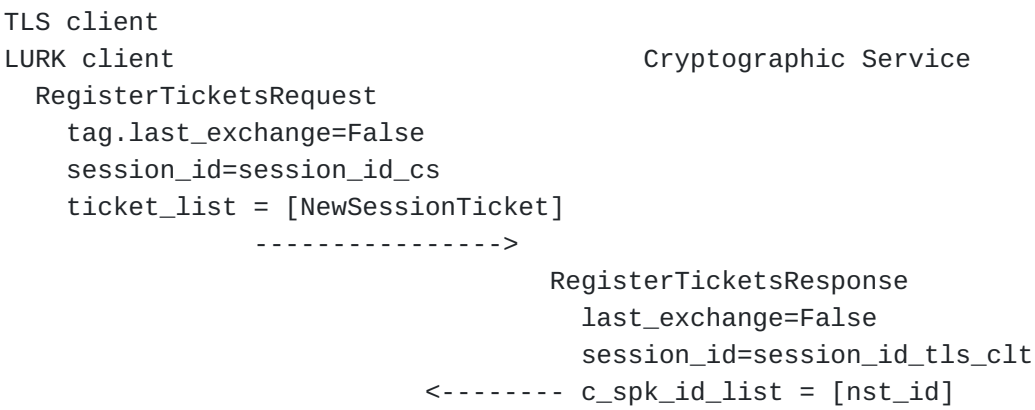
```
TLS client
LURK client                                  Cryptographic Service
  CHandAndAppSecretRequest
    tag.last_exchange=False
    session_id=session_id_cs
    ephemeral
      method = no_secret
    handshake = ServerHello, {EncryptedExtensions}...,{Finished}.
    secret_request = h_c, h_s, a_c, a_s ------->
                                    CHandAndAppSecretResponse
                                      tag
                                        last_exchange=False
                                         cert_request=False
                                      session_id=session_id_tls_clt
                                      ephemeral_list = []
                                      secret_request = h_s, h_c, a_s, and
```

Upon reception of the response, the TLS client generates the necessary keys to decrypt and encrypt the handshake message and terminates the TLS handshake. The TLS client is also able to decrypt and encrypt application traffic.

In this section, we assume that after some time, the TLS client receives a NewSessionTicket from the TLS server. The TLS client will then transmit the NewSessionTicket to CS so that it can generate the associated PSK that will be used for the authentication.

As multiple NewSessionTickets may be sent, in this example, both TLS client and CS enable further additional registrations by unsetting tag.last_exchange. For each registered NewSessionTicket, CS returns c_spk_id that will use for further references. The c_spk_ids are managed by CS which can ensure the uniqueness of these references as opposed to using the ticket field that is assigned by the TLS server.

Appendix A.5 illustrates how session resumption is performed using PSK / PSK-ECDHE authentication.



A.5. TLS client unauthenticated PSK / PSK-ECDHE

This section describes the intercation between a TLS client and a CS for a PSK-ECDHE TLS handshake. Appendix A.4 shows how the PSK may be provisioned during a ECDHE TLS handshake. The scenario described in this section presents a number of similarities to the one described in Appendix A.4. As such, we expect the reader to be familiar with Appendix A.4 and will highlight the differences with Appendix A.4 to avoid to repeat the description.

In this section, the PSK is protected by the CS, but the (EC)DHE private keys are generated by the TLS client and as such are considered as unprotected. As the (EC)DHE secret are generated by the TLS client, the method is set to no_secret, and the key_share extension is fully provided in the ClientHello. However, the ClientHello do not carry the PreSharedKeyExtension. Instead, this extension is built from the NewSessionTicket identifier nst_id provided in our case from a previous c_register_ticktes exchange (see Appendix A.4 }. The TLS client requests the binder_key associated to nst_id in order to be able to complete the binders.

Upon receiving the message, the CS, computes the binder_keys, complete the ClientHello in order to synchronize its TLS handshake with the TLS client (and the TLS server). As CS does not generate any (EC)DHE, the ephemeral_list is empty.

```
TLS client
LURK client                                  Cryptographic Service
  CInitClientHello
    session_id = session_id_tls_client
    freshness = sha256
    ephemeral
      method = no_secret
    handshake = ClientHello without PreSharedKeyExtension
    c_psk_id_list = [nst_id]
    secret_request = [b]    ------>
                                    CInitClientHello
                                      session_id=session_id_cs
                                      ephemeral_list = []
                                      secret_list=[binder_key]
```

When the TLS client receives the responses from the TLS server, the handshake and application secrets are requested with a c_hand_and_app similarly to Appendix A.4. The only difference here is that (EC)DHE have been generated by the TLS client and the shared secret needs to be provided to CS as described below:

```
TLS client
LURK client                                  Cryptographic Service
  CHandAndAppSecretRequest
    tag.last_exchange=False
    session_id=session_id_cs
    ephemeral
      method = e_generated
      shared_secret
    handshake = ServerHello, {EncryptedExtensions}...,{Finished}.
    secret_request = h_c, h_s, a_c, a_s ------->
                                    CHandAndAppSecretResponse
                                      tag
                                        last_exchange=False
                                         cert_request=False
                                      session_id=session_id_tls_clt
                                      ephemeral_list = []
                                      secret_request = h_s, h_c, a_s, and
```

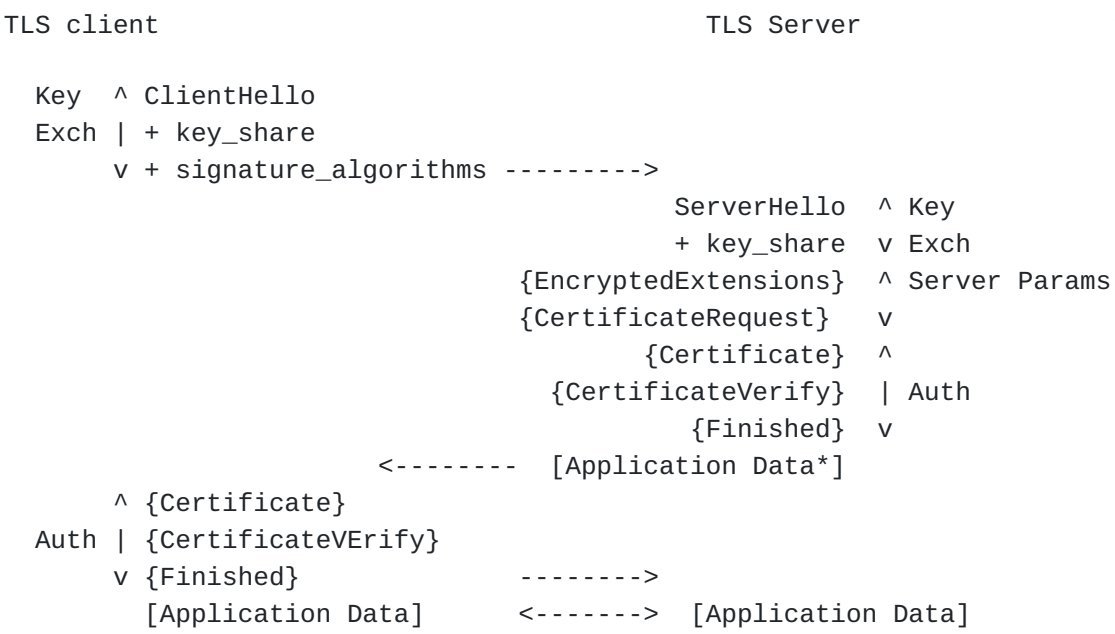
Upon receiving the response, the TLS client proceeds similarly to the TLS client described in Appendix A.4.

A.6. TLS client authenticated ECDHE

This section provides scenarios when the TLS client is authenticated during the TLS handshake. Post handshake authentication is detailed in Appendix A.7

A.6.1. (EC)DHE or Proposed PSK protected by the CS

When the (EC)DHE part have been generated by the CS, or the proposed PSK are protected by the CS, the TLS client sends a ClientHello after a c_client_hello exchange with CS (see Appendix A.5 or Appendix A.4). The request for TLS client authentication is indicated by a encrypted CertificateRequest sent by the TLS server as indicated below:



The TLS client is unaware of the presence of the CertifcateRequest until it has decrypted the message with a key derived from the handshake secrets. As a result, the TLS client initiates a c_hand_an_app_secret exchange as described in Appendix A.5 or Appendix A.4.

CS proceeds as described in Appendix A.5 or Appendix A.4. However, after the messages have been decrypted, CS proceeds to the generation of the signature and returns the necessary information to build the CertificateVerify. CS indicates their presence by setting tag.cert_request and returns the certificate, the sig_algo and sig as described below:

```
TLS client
LURK client                                  Cryptographic Service
  CHandAndAppSecretRequest
    tag.last_exchange=False
    session_id=session_id_cs
    ephemeral
      method = e_generated
      shared_secret
    handshake = ServerHello, {EncryptedExtensions}...,{Finished}.
    secret_request = h_c, h_s, a_c, a_s ------->
                                    CHandAndAppSecretResponse
                                      tag
                                        last_exchange=False
                                        cert_request=True
                                      session_id=session_id_tls_clt
                                      ephemeral_list = []
                                      secret_request = h_s, h_c, a_s, and
                                      certificate
                                        certificate_type = finger_print
                                      sig_algo = ed25519
                                      sig
```

Note that in the example above, (EC)DHE have not been generated by the CS, but the c_client_hello was motivated to propose a protected PSK. As the PSK has not been agreed for authentication by the TLS server, the TLS session does not provide PFS and the protection is similar as the one described in {sec:ex:clt:auth:ecdhe-certverify}, where the TLS client would have proposed directly ECDHE with (EC)DHE generated by the TLS client.

A.6.2. (EC)DHE provided by the TLS client

This section considers a TLS client that proposes to authenticate the TLS server using ECDHE with (EC)DHE private parts being generated by the TLS client.

The TLS client does not need to interact with CS to build its ClientHello. Similarly, as the (EC)DHE private part have been generated by the TLS client, the TLS client is able to perform the key schedule and derive the necessary keys to decrypt the encrypted response from the TLS server. Upon receiving a CertificateRequest, the TLS client requests CS to generate the signature needed to send the CertificateVerify. The exchange is very similar as the one s_init_cert_verify (see Appendix A.1.2). As the (EC)DHE shared secret is generated by the TLS client, the method is necessarily set to e_generated. The handshake is set to the ClientHello ... server Finished, and the certificate carries the reference to the TLS client certificate, so CS picks the appropriated private key. sig_algo designates the signature algorithm.

```
TLS server
LURK client                                  Cryptographic Service
  CInitClientFinishedRequest
    tag.last_exchange=True   -------->
    freshness = sha256
    ephemeral
      method = e_generated
      key
        group = x25519
        shared_secret = shared_secret
    handshake = hanshake
    certificate
      certificate_type = finger_print
    sig_algo = ed25519
                                    CInitClientFinishedResponse
                                      tag.last_exchange=True
                                      signature = sig
                                    <---------
```

A.7. TLS client authenticated - post handshake authentication

Post handshake authentication may be requested at any time after the TLS handshake is completed as long as the TLS client has indicated its support with a post_handshake_authentication extension.

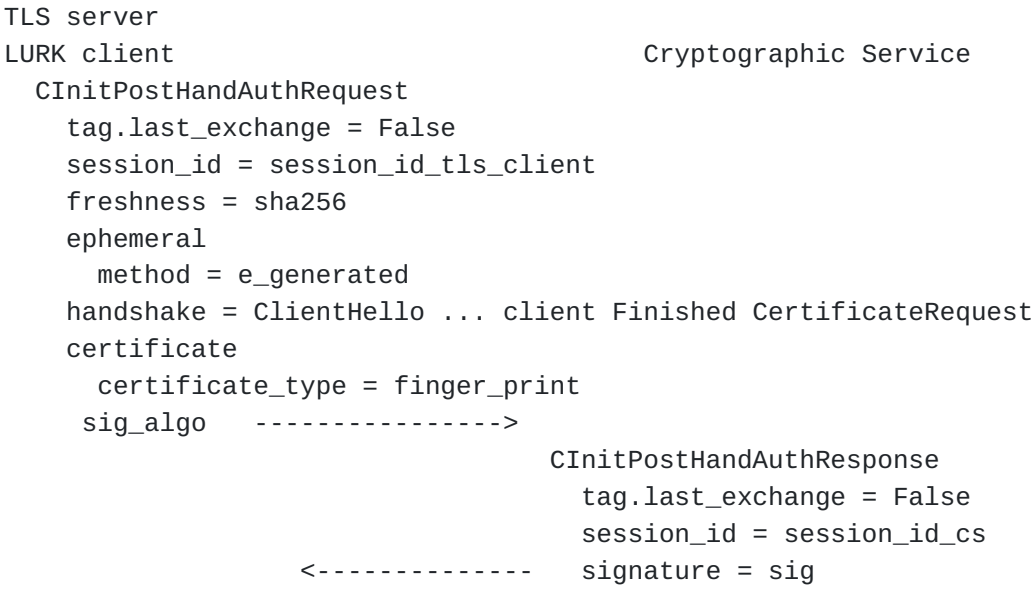
If the establishment of the TLS session did not required any interactions with the CS, post handshake authentication is performed with a c_init_post_hand_auth exchange as described in Appendix A. 7.1. When the TLS handshake already required some interactions with CS the post handshake authentication is performed using a c_post_hand_auth described in {sec:ex:clt:auth:post_continued}.

In some cases, both c_init_post_hand_auth and c_post_hand_auth can be used. When this is possible, c_post_hand_auth is preferred as the handshake context is already being provisioned in the CS. On the other hand, when the shared secret is only known to the CS, c_init_post_hand_auth cannot be used instead.

A.7.1. Initial Post Handshake Authentication

This situation describes the case where the TLS client has performed the TLS handshake without interacting with the CS. As a result, if involved PSK, (EC)DHE shared secrets are unprotected and hosted by the TLS client. Upon receiving a CertificateRequest, the TLS client sends session_id and freshness to initiate the LURK session. tag.last_exchange is set in order to accept future post handshake authentication request. method is set to secret_provide as CS is unable to generate the (EC)DHE shared secret. handshake is set to the full handshake including the just received CertificateRequest

message. The certificate represents the TLS client certificate to determine the private key involved in computing the signature. sig_algo specifies the signature algorithm.



A.7.2. Post Handshake Authentication

In this scenario, the post authentication is performed while a LURK session has already been set. Upon receiving the CertificateRequest, the TLS client proceeds similarly to the initial post handshake authentication as described in As a result, the exchange is illustrated below:Appendix A.7.1 except that the LURK session does not need to be initiated, the shared secret is already known to CS and the handshake is only constituted of the remaining CertificateRequest message.

Author's Address

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

Email: daniel.migault@ericsson.com