

LURK  
Internet-Draft  
Intended status: Standards Track  
Expires: 27 January 2022

D. Migault  
Ericsson  
26 July 2021

LURK Extension version 1 for (D)TLS 1.3 Authentication  
draft-mglt-lurk-tls13-05

## Abstract

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

## Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 27 January 2022.

## Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the [Trust Legal Provisions](#) and are provided without warranty as described in the Simplified BSD License.

Internet-Draft

LURK/TLS 1.3

July 2021

## Table of Contents

<a href="#">1.</a>	<a href="#">TODO</a>	<a href="#">3</a>
<a href="#">2.</a>	<a href="#">Introduction</a>	<a href="#">3</a>
<a href="#">3.</a>	<a href="#">Terminology</a>	<a href="#">4</a>
<a href="#">4.</a>	<a href="#">LURK Header</a>	<a href="#">6</a>
<a href="#">5.</a>	<a href="#">Structures</a>	<a href="#">7</a>
<a href="#">5.1.</a>	<a href="#">secret_request</a>	<a href="#">7</a>
<a href="#">5.2.</a>	<a href="#">handshake</a>	<a href="#">8</a>
<a href="#">5.3.</a>	<a href="#">session_id</a>	<a href="#">10</a>
<a href="#">5.4.</a>	<a href="#">freshness</a>	<a href="#">11</a>
<a href="#">5.5.</a>	<a href="#">ephemeral</a>	<a href="#">13</a>
<a href="#">5.5.1.</a>	<a href="#">shared_secret_provided:</a>	<a href="#">13</a>
<a href="#">5.5.2.</a>	<a href="#">secret_generated:</a>	<a href="#">13</a>
<a href="#">5.5.3.</a>	<a href="#">no_secret</a>	<a href="#">14</a>
<a href="#">5.6.</a>	<a href="#">selected_identity</a>	<a href="#">15</a>
<a href="#">5.7.</a>	<a href="#">certificate</a>	<a href="#">16</a>
<a href="#">5.8.</a>	<a href="#">tag</a>	<a href="#">18</a>
<a href="#">5.9.</a>	<a href="#">secret</a>	<a href="#">19</a>
<a href="#">5.10.</a>	<a href="#">signature</a>	<a href="#">20</a>
<a href="#">6.</a>	<a href="#">LURK exchange on the TLS server</a>	<a href="#">20</a>
<a href="#">6.1.</a>	<a href="#">s_init_cert_verify</a>	<a href="#">20</a>
<a href="#">6.2.</a>	<a href="#">s_new_tickets</a>	<a href="#">21</a>
<a href="#">6.3.</a>	<a href="#">s_init_early_secret</a>	<a href="#">22</a>
<a href="#">6.4.</a>	<a href="#">s_hand_and_app_secret</a>	<a href="#">23</a>
<a href="#">7.</a>	<a href="#">LURK exchange on the TLS client</a>	<a href="#">24</a>
<a href="#">7.1.</a>	<a href="#">c_init_post_hand_auth</a>	<a href="#">26</a>
<a href="#">7.2.</a>	<a href="#">c_post_hand_auth</a>	<a href="#">28</a>
<a href="#">7.3.</a>	<a href="#">c_init_cert_verify</a>	<a href="#">28</a>
<a href="#">7.4.</a>	<a href="#">c_init_client_hello</a>	<a href="#">30</a>
<a href="#">7.5.</a>	<a href="#">c_client_hello</a>	<a href="#">32</a>
<a href="#">7.6.</a>	<a href="#">c_hand_and_app_secret</a>	<a href="#">33</a>
<a href="#">7.7.</a>	<a href="#">c_register_tickets</a>	<a href="#">35</a>
<a href="#">8.</a>	<a href="#">Security Considerations</a>	<a href="#">35</a>
<a href="#">9.</a>	<a href="#">IANA Considerations</a>	<a href="#">36</a>
<a href="#">10.</a>	<a href="#">Acknowledgments</a>	<a href="#">37</a>
<a href="#">11.</a>	<a href="#">References</a>	<a href="#">37</a>
<a href="#">11.1.</a>	<a href="#">Normative References</a>	<a href="#">37</a>
<a href="#">11.2.</a>	<a href="#">Informative References</a>	<a href="#">37</a>
<a href="#">Appendix A.</a>	<a href="#">Annex</a>	<a href="#">37</a>
<a href="#">A.1.</a>	<a href="#">TLS server ECDHE (no session resumption)</a>	<a href="#">37</a>
<a href="#">A.1.1.</a>	<a href="#">ecdhe generated on the CS</a>	<a href="#">39</a>
<a href="#">A.1.2.</a>	<a href="#">ecdhe generated by the TLS server</a>	<a href="#">39</a>

<a href="#">A.2.</a>	TLS server ECDHE ( with session resumption ) . . . . .	<a href="#">40</a>
<a href="#">A.3.</a>	TLS server PSK / PSK-ECDHE . . . . .	<a href="#">42</a>
<a href="#">A.4.</a>	TLS client unauthenticated ECDHE . . . . .	<a href="#">45</a>
<a href="#">A.5.</a>	TLS client unauthenticated PSK / PSK-ECDHE . . . . .	<a href="#">49</a>
<a href="#">A.6.</a>	TLS client authenticated ECDHE . . . . .	<a href="#">50</a>

<a href="#">A.6.1.</a>	(EC)DHE or Proposed PSK protected by the CS . . . . .	<a href="#">51</a>
<a href="#">A.6.2.</a>	(EC)DHE provided by the TLS client . . . . .	<a href="#">52</a>
A.7.	TLS client authenticated - post handshake authentication . . . . .	<a href="#">53</a>
<a href="#">A.7.1.</a>	Initial Post Handshake Authentication . . . . .	<a href="#">54</a>
<a href="#">A.7.2.</a>	Post Handshake Authentication . . . . .	<a href="#">54</a>
Author's Address	. . . . .	<a href="#">55</a>

## [1.](#) TODO

1. check the terminology is used. PSK agreed....
2. move the handshake description to the s\_exchange description
3. state diagram for the server.

## [2.](#) Introduction

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

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

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

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

The current document extends the scope of the LURK extension for TLS

1.2 in that it defines the CS on the TLS server as well as on the TLS client and the CS can operate in non delegating scenarios.

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

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

Role	LURK exchange	secret	sign	ticket
server	s_init_early_secret	yes	-	-
server	s_init_cert_verify	yes	yes	-
server	s_hand_and_app_secret	yes	-	-
server	s_new_ticket	yes	-	yes
client	c_init_post_hand_auth	-	yes	-
client	c_post_hand_auth	-	yes	-
client	c_init_cert_verify	yes	yes	-
client	c_init_early_secret	yes	-	-
client	c_init_hand_secret	yes	-	-
client	c_hand_and_app_secret	yes	-	-
client	c_register_tickets	yes	-	yes

Figure 1: Operation associated to LURK exchange

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

client and by the CS.

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

### 3. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [BCP 14](#) [[RFC2119](#)] [[RFC8174](#)] when, and only when, they appear in all capitals, as shown here.

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

ECHDE designates the ECDHE authentication defined in [[RFC8446](#)].  
(EC)DHE

designates the shared secret agreed by the key\_share extension ([section 4.2.8 of \[RFC8446\]](#)) during a TLS handshake.

**PSK Proposed** A TLS handshake between a TLS client and a TLS server is said to be "PSK proposed" when 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. A TLS client is said to "propose PSK" when its TLS handshake is PSK proposed.

**PSK Agreed** A TLS handshake between a TLS client and a TLS server is said to be "PSK agreed" when the TLS handshake is PSK proposed and the ServerHello contains a psk\_key\_exchange\_modes and a pre\_shared\_key extension. A TLS client and a TLS server are said to have "agreed on PSK" when its TLS handshake is PSK agreed.

**ECDHE Agreed** A TLS handshake between a TLS client and a TLS server is said to be "ECDHE agreed" when the ServerHello contains neither a psk\_key\_exchange\_modes or a pre\_shared\_key extension. As currently TLS proposes only ECDHE and PSK based authentication, when PSK agreed is false, ECDHE agreed is true. A TLS client and a TLS server are said to have "agreed on ECDHE" when its TLS

handshake is PSK agreed.

**Key Share Proposed** A TLS handshake between a TLS client and a TLS server is said to be "key shared proposed" or "(EC)DHE proposed" when the latest ClientHello contains a key\_share extension ([section 4.2.8 of \[RFC8446\]](#)). A TLS client is said to "propose PSK" when its TLS handshake is PSK proposed.

**Key Share Agreed** A TLS handshake between a TLS client and a TLS server is said to be "key shared agreed" or "(EC)DHE agreed" when the TLS handshake is key shared proposed and the ServerHello contains neither a key\_share extension. A TLS client and a TLS server are said to have "agreed on (EC)DHE" when its TLS handshake is key share agreed.

**Early Data Enabled** A TLS client is said to "support early data" or "enable early data" when its latest ClientHello contains a early\_data extension ([section 4.2.10 of \[RFC8446\]](#)).

**Post Handshake Enabled:** A TLS client is said to "support early data" or "enable early data" when its latest ClientHello contains a post\_handshake\_auth extension.

#### [4.](#) LURK Header

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

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

```
enum {  
    capabilities(0),  
    ping(1),
```

```

    s_init_cert_verify(2),
    s_new_ticket(3),
    s_init_early_secret(4),
    s_hand_and_app_secret(5),
    c_binder_key(6),
    c_init_early_secret(7),
    c_init_hand_secret(8),
    c_hand_secret(9),
    c_app_secret(10),
    c_cert_verify(11),
    c_register_tickets(12),
    c_post_hand(13), (255)
}TLS13Type;

```

```

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

    invalid_psk
    invalid_freshness

    invalid_request
    invalid_key_id_type
    invalid_key_id
    invalid_signature_scheme
    invalid_certificate_type
    invalid_certificate
    invalid_certificate_verify
    invalid_secret_request
    invalid_handshake

```

```

    invalid_extension
    invalid_ephemeral
    invalid_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;
        uint32 length;
    } LURKHeader;

```

## 5. Structures

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

### 5.1. secret\_request

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

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



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

Figure 2: secret\_request structure

LURK exchange	Permitted secrets
s_init_cert_verify	h_c*, h_s*, a_c*, a_s*, x*
s_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_post_hand_auth	-
c_post_hand_auth	-
c_init_cert_verify	a_c*, a_s*, x*
c_init_client_hello	b*, e_c*, e_x*
c_client_hello	b*, e_c*, e_x*
c_hand_and_app_secret	h_c, h_s, a_c*, a_s*, x*, r*
c_register_tickets	-

\* indicates the secret MAY be requested

- indicates no secrets can be requested

Figure 3: secret\_request permitted values per LURK exchange

## 5.2. handshake

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

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

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

On the TLS server:

- \* (a) `ServerHello.random` value provided by the LURK client requires specific treatment as described in [Section 5.4](#) before being inserted in the TLS handshake variable.
- \* (b) When the shared secret ( and so the private ECDHE ) is generated by the CS, the `KeyShareServerHello` structure cannot be provided to the CS by the LURK client in a `ServerHello` and is instead completed by the CS as described in [Section 5.5](#).
- \* (c) The TLS Certificate structure MUST not be provided by the LURK client as part of the handshake structure. Instead, the CS generates the Certificate message from the certificate structure described in [Section 5.7](#). The handshake MUST NOT contain a TLS Certificate message and CS SHOULD raise an `invalid_handshake_error` if such message is found in the TLS handshake. When a client Certificate is provided, the CS SHOULD raise an `invalid_handshake_error` in the absence of a `CertificateRequest` message.
- \* (d) The Certificate and Finished messages are not provided in a handshake structure by the LURK client but are instead generated by the CS as described in [Section 5.10](#).
- \* (e) Some authentication PSK\_ECHDE or ECDHE requires the agreement of a shared ECDHE secret. This is indicated by the presence of `key_share` extension in both `ClientHello` and `ServerHello`. When these extensions are not found, the CS SHOULD raise an error. Note that in the case of PSK / PSK-ECDHE, the presence or absence of `key_share` extension MAY be used to distinguish between the two authentication methods.
- \* (f) ECDHE authentication does not involve the agreement of a PSK. This is indicated by the presence of a `key_share` extension in both `ClientHello` and `ServerHello`. When these extensions are found, the CS SHOULD raise an error.

Internet-Draft

LURK/TLS 1.3

July 2021

- \* (g) PSK and PSK\_ECDHE requires the agreement of a PSK, so a psk is expected in the ClientHello as well as - when present in the ServerHello. When this extension are not found, the CS SHOULD raise an error.

LURK exchange	TLS handshake	CS operations
s_init_cert_verify	ClientHello ... later of server EncryptedExtensions / CertificateRequest	a,b,c,d,e,f
s_new_ticket	earlier of client Certificate / client CertificateVerify / Finished ... Finished	c
s_init_early_secret	ClientHello	a, g
s_hand_and_app_secret	ServerHello ... later of server EncryptedExtensions / CertificateRequest	b, g
c_init_post_hand_auth	ClientHello ... ServerHello CertificateRequest	e
c_post_hand_auth	CertificateRequest	
c_init_cert_verify	ClientHello...server Finished	e,f
c_init_client_hello	(Partial) ClientHello or ClientHello, HelloRetryRequest, (Partial) ClientHello	
c_client_hello	HelloRetryRequest, (Partial) ClientHello	
c_hand_and_app_secret	ServerHello, {EncryptedExtensions} ... later of { server Finished } / EndOfEarlyData	
c_register_tickets	-	

Figure 4: handshake values per LURK exchange

Handshake handshake<0..2^32> //RFC8446 [section 4](#) (clear) clientHello...client f

### [5.3.](#) session\_id

The session\_id is a 32 bit identifier that identifies a LURK session

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

Table Figure 5 indicates the presence of the `session_id`.

LURK exchange	session_id
s_init_cert_verify	*
s_new_ticket	y
s_init_early_secret	y
s_hand_and_app_secret	y
c_init_post_hand_auth	*
c_post_hand_auth	y
c_init_cert_verify	*
c_init_client_hello	y
c_client_hello	y
c_hand_and_app_secret	y
c_register_tickets	y

y indicates the `session_id` is present

- indicates `session_id` may be absent

\* indicates `session_id` may be present (depending on the `tag.last_message`)

Figure 5: `session_id` in LURK exchanges

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

#### 5.4. freshness

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

using the freshness function and the `ServerHello.random` provided by the LURK client in the handshake structure. The CS operates similarly on the TLS client and generates the `ClientHello.random` of the TLS handshake using the freshness function as well as the `ClientHello.random` value provided by the LURK client in the handshake structure.

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

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

LURK exchange	freshness
s_init_cert_verify	y
s_new_ticket	-
s_init_early_secret	-
s_hand_and_app_secret	y
c_init_post_hand_auth	y
c_post_hand_auth	-
c_init_cert_verify	y
c_init_client_hello	y
c_client_hello	-
c_hand_and_app_secret	-
c_register_tickets	-

y indicates freshness is present

- indicates freshness is absent

Figure 6: freshness in LURK exchange

The extension data is defined as follows:

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

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

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

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

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

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

## [5.5.](#) ephemeral

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

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

### [5.5.1.](#) `shared_secret_provided`:

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

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

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

#### [5.5.2.](#) secret\_generated:

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

The CS MUST check the group field in the KeyShareServerHello, and get the public value of the TLS client from the KeyShareClientHello. The CS performs the same checks as described in [\[RFC8446\] section 4.2.8](#).

The CS generates the private and public (EC)DH keys, computes the shared key and return the KeyShareEntry server\_share structure defined in [\[RFC8446\] section 4.2.8](#) to the LURK client.

#### [5.5.3.](#) no\_secret

With PSK authentication, (EC)DHE keys and shared secrets are not needed. The CS SHOULD check the PSK authentication has been agreed, that is pre\_shared\_key and psk\_key\_exchange\_modes extensions are not present in the ClientHello and in the ServerHello

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

LURK exchange	ephemeral	ephemeral_method= secret		
		no	provided	generated
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_post_hand	y	-	y	-
c_post_hand	y	-	y	y
c_init_cert_verify	y	-	y	y
c_init_client_hello	y	y	-	y
c_client_hello	y	y	-	y
c_hand_and_app_secret	y	y	y	-
c_register_tickets	-	-	-	-

y indicates presence of ephemeral or possible value for `ephemeral_method`  
 - indicates absent or ephemeral or incompatible value for `ephemeral_method`

Figure 7: Ephemeral field in LURK exchange

The extension data is defined as follows:

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

```
EphemeralRequest {
    EphemeralMethod method;
```



```

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

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

```

#### 5.6. selected\_identity

The selected\_identity indicates the identity of the PSK used in the key schedule. The selected\_identity is expressed as a (0-based) index into the identities in the client's list. The client's list is provided in the pre\_shared\_key extension as expressed in [\[RFC8446\]](#) [section 4.2.11](#).

The LURK client MUST provide the selected\_identity only when PSK or PSK-authentication is envisioned and when the PSK has not been provided earlier. These exchanges are s\_init\_early\_secret on the TLS server and c\_init\_early\_secret and c\_init\_hand\_secret on the TLS client side.

+-----+-----+	
LURK exchange	req
+-----+-----+	
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_cert_verify	-
c_init_client_hello	-
c_client_hello	-
c_hand_and_app_secret	-
c_register_tickets	-
+-----+-----+	

y indicates the selected\_identity is present  
 - indicates the selected\_identity is absent

Figure 8: psk\_id in LURK exchange

The extension data is defined as follows:

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

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

## [5.7.](#) certificate

The certificate structure indicates the presence and associated value of the Certificate message in the TLS handshake.

Upon receiving a certificate field, the CS MUST: 1. ensure coherent with the handshake messages - typically authentication method is ECDHE and not PSK or PSK-ECDHE. 2. ensure the provided value corresponds to an acceptable provisioned value. 3. generate the appropriated corresponding message.

If the CS is not able to understand the lurk\_tls13\_certificate field, it SHOULD return an invalid\_certificate error.

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

LURK exchange	certificate	certificate type
s_init_cert_verify	y	server certificate
s_new_ticket	*	client certificate
s_init_early_secret	-	
s_hand_and_app_secret	-	
c_init_post_hand_auth	y	client_certificate
c_post_hand_auth	y	client_certificate
c_init_cert_verify	y	client certificate
c_init_client_hello	-	
c_client_hello	-	
c_hand_and_app_secret	y	client_certificate
c_register_tickets	-	

\* indicates certificate type MAY be set to empty.

y indicates certificate type MUST NOT be set to empty

- indicates the certificate structure is absent

Figure 9: certificate per LURK exchange

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

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

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

empty indicates there is no certificates provided by this field.

fingerprint a 4 bytes finger print length that represents the

fingerprinting of the TLS Certificate message. Fingerprinting is described in [[RFC7924](#)] and takes as input the full handshake message - that is a message of message type certificate with that contain the Certificate as its message\_data. In this document only the 4 most left bytes of the output are considered.  
uncompressed

indicates the Certificate message as defined in [[RFC8446](#)] is provided.

#### [5.8.](#) tag

This field provides extra information. Currently, the tag structure defines tag.last\_message and tag.cert\_request.

The LURK client or the CS sets the tag.last\_message to terminate the LURK session. The tag.cert\_request is only used by the CS in a c\_hand\_and\_app\_secret exchange. The tag.cert\_request by the CS when a CertificateRequest has been found in the handshake and that the CS includes in its response the necessary information to the TLS client to build a CertificateVerify message (see [Section 7.6](#) ).

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	cert_request
2-7	RESERVED

Figure 10: tag description

LURK exchange	last_message	cert_request
s_init_cert_verify	y	-
s_new_ticket	y	-
s_init_early_secret	-	-
s_hand_and_app_secret	y	-
c_init_post_hand_auth	y	-
c_post_hand_auth	y	-
c_init_cert_verify	y	-
c_init_client_hello	-	-
c_client_hello	-	-
c_hand_and_app_secret	y	y (response)
c_register_tickets	y	-

y indicates tag is present  
 - indicates tag is absent

Figure 11: tag per LURK exchange

## 5.9. secret

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

```

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

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

```

secret\_type: The type of the secret or key

secret\_data: The value of the secret.

## 5.10. signature

The signature requires the signature scheme, a private key and the appropriated context. The signature scheme is provided using the SignatureScheme structure defined in [\[RFC8446\] section 4.2.3](#), the private key is derived from the lurk\_tls13\_certificate [Section 5.7](#) and the context is derived from the handshake [Section 5.2](#) and lurk\_tls13\_certificate [Section 5.7](#).

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

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

+-----+-----+

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

## [6.](#) LURK exchange on the TLS server

This section describes the LURK exchanges that are performed on the TLS server. Unless specified used structures are described in [Section 5](#)

### [6.1.](#) s\_init\_cert\_verify

s\_init\_cert\_verify initiates a LURK session when the server is authenticated with ECDHE. The ClientHello received by the TLS server, and the ServerHello and optionally the HelloRetryRequest MUST carry a key\_share extension.

If the LURK client is configured to not proceed to further exchange, it sets the last\_exchange bit of the tag. When this bit is set, the session\_id is ignored. The CS sets the last\_exchange bit if the last\_exchange bit has been set by the LURK client or when it has been configured to not accept further LURK exchanges, such as s\_new\_ticket.

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

```

}SInitCertVerifyRequest

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

```

sig\_algo SignatureScheme is defined in [\[RFC8446\] section 4.2.3](#).

## [6.2](#). 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 the CS to the TLS server. In fact, if the LURK client requests and receives the resumption\_master\_secret it is able to emit on its own NewSessionTicket. As a result s\_new\_ticket LURK exchanges are only initiated if the TLS server expects to perform session resumption and the CS responds only if session\_resumption is enabled.

The CS MAY responds with a resumption\_master\_secret based on its policies.

The LURK client MAY perform multiple s\_new\_ticket exchanges. The LURK client and CS are expected to advertise by setting the last\_exchange bit in the tag field.

```

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

```



```

    uint8 ticket_nbr;
    uint16 secret_request;
} SNewTicketRequest;

```

```

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

```

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

### [6.3.](#) s\_init\_early\_secret

s\_init\_early\_secret initiates a LURK session when the server is authenticated by the PSK or PSK-ECDHE methods. This means the ClientHello received by the TLS server and ServerHello responded by the TLS server MUST carry the pre\_shared\_key and psk\_key\_exchange\_modes extensions.

selected\_identity indicates the selected PSK

```

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

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

```

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

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

#### [6.4.](#) `s_hand_and_app_secret`

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

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

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

Internet-Draft

LURK/TLS 1.3

July 2021

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

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

## [7.](#) LURK exchange on the TLS client

The CS described in this document considers the case where PSK and (EC)DHE private part MAY be protected by the CS. This document does not consider the case where a TLS client may propose a combination of protected and unprotected PSKs. Either all proposed PSK are unprotected by the CS or all PSK are protected by the CS. Similarly, this document does not consider the case where a TLS client may propose a combination of protected and unprotected (EC)DHE. Either all (EC)DHE are generated by the CS or all (EC)DHE are generated by the TLS client.

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

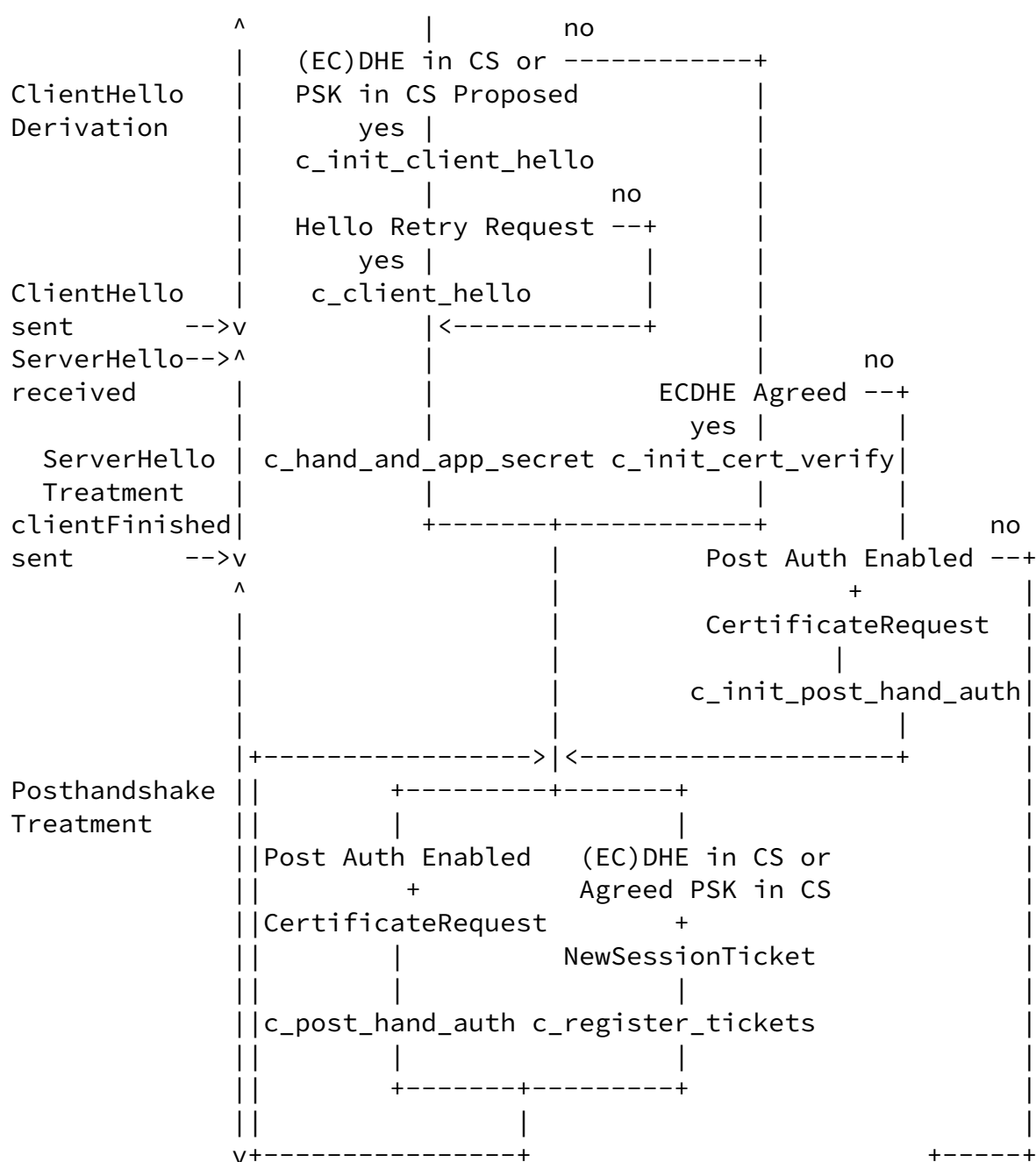




Figure 12: LURK client State Diagram

The TLS client needs to interact with the CS when either the TLS client proposes PSKs and the CS hosts the PSKs or when the TLS client requests the CS to generate the (EC)DHE private key. The TLS client requests with a `c_init_client_hello` exchange the CS the `binder_keys`, the (EC)DHE public part as well as early secrets and is thus able to start the TLS handshake, that is sending a `ClientHello` with potential early data. In case a `HeloRetryRequest` is received, the TLS client

pursue with a `c_client_hello` exchange to complete the second `ClientHello`. Upon receiving a `ServerHello` as well as other messages by the TLS server, the TLS client requests via a `c_hand_and_App_secret` exchange the necessary information (signature, secrets) to complete the TLS handshake.

When the TLS client does not propose PSKs that are protected by the CS nor does request the CS to generate the private part of the (EC)DHE shared secret, the TLS client does not need to interact with the CS to send its `ClientHello`. However, the TLS client may interact with the CS to authenticate itself to the TLS server via the generation of a signature contained in a `CertificateVerify` message. Such authentication can be performed during the TLS handshake when ECDHE is agreed and when the TLS client enables post handshake authentication, in which case the signature is requested via a `c_init_cert_verify` exchange.

Once the TLS handshake is completed, and the TLS client has enabled post handshake authentication, the TLS server may request by sending a `CertificateRequest` the TLS client to authenticate itself with a signature. If the TLS client has already started a LURK session associated to the TLS handshake the signature is requested via a `c_post_hand_auth` exchange. Otherwise, the signature is requested via a `c_init_post_hand_auth` exchange. Multiple post hand shake authentication may be performed in both cases and additional signature generation are requested via `c_post_hand_auth` exchange.

Similarly, once the TLS handshake is completed, the TLS client may

receive NewSessionTickets from the TLS server to perform session resumption. If the TLS client has its (EC)DHE or the PSK in use protected by the CS - the NewSessionTicket is registered via a c\_register\_ticket. Multiple NewSessionTickets may be registered. When no protected PSK have been agreed and (EC)DHE are not generated by the CS, the TLS client may generate the PSK session resumption. As a result, it cannot be registered in the CS as to prevent providing a false sense of security.

### [7.1.](#) c\_init\_post\_hand\_auth

The c\_init\_post\_hand\_auth occurs when the TLS client performs post handshake authentication while no previous interactions occurred between the TLS client and the CS. More specifically, the (EC)DHE shared secrets have been generated by the TLS client, and the proposed keys, if proposed previously by the TLS client, are not protected by the CS.

The CS completes and returns the signature as described in [Section 5.10](#).

Since session resumption secrets are not protected by the CS, the registration of NewSessionTickets is not expected, and the only exchanges that MAY follow are additional post handshake authentications described in [Section 7.2](#).

```
struct{
    uint8 tag;
    select (tag.last_exchange){
        case False:
            uint32 session_id;
    }
    FreshnessFunc freshness;
    Ephemeral = ephemeral      ## ephemeral_method = secret_provided
    Handshake handshake<0..2^32> //RFC8446 section 4
    LURKTLS13Certificate cert;
    SignatureScheme sig_algo;
}CInitPostHandAuthRequest
```

```
struct{
    unit8 tag
```

```

select (tag.last_exchange){
  case False:
    uint32 session_id;
  }
  Signature signature
}CInitPostHandAuth

```

`tag` is defined in [Section 5.8](#). The TLS client sets `tag.last_message` if further post handshake authentications are expected. Similarly the CS sets the `tag.last_message` if further post handshake authentications are permitted. Note that it is out of scope of this specification to specify the reasons, but it is RECOMMENDED the CS sets life time to LURK session as well as limits the maximum number of post handshake authentications.

`ephemeral` is defined in [Section 5.5](#) and `ephemeral_method` MUST be set to 'secret\_provided' when the TLS handshake agreed an ECDHE or a PSK-ECDHE authentication. The `ephemeral_method` MUST be set to 'no\_secret' when PSK the TLS handshake agreed on PSK authentication.

`handshake` is defined in [Section 5.2](#) and post handshake authentication MUST be enabled by the TLS client.

`cert` is defined in [Section 5.7](#) and is used by the TLS client to

indicate the expected certificate to be used to compute the signature as well as the certificate that is expected to be send further to the TLS server.

`sig_algo` is defined in [Section 5.10](#) and indicates the selected algorithm.

`signature` is defined in [Section 5.10](#).

## [7.2](#). `c_post_hand_auth`

The `c_post_hand_auth` exchange enables a TLS client to perform post handshake authentication.

This exchange is followed by a `c_register_ticket` or a `c_post_hand_and_app` exchange.

```
struct{
    Tag tag
    uint32 session_id
    Ephemeral = ephemeral      ## ephemeral_method = secret_provided
    Handshake handshake<0.. $2^{32}$ > // CertificateRequest
    LURKTLS13Certificate cert;
    SignatureScheme sig_algo;
}CPostHandAuthRequest
```

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

tag, sig\_algo, cert, signature are described in [Section 7.1](#)

handshake is defined in [Section 5.2](#) and is only composed of a CertificateRequest. However, post handshake authentication MUST be enabled by the TLS client.

### [7.3.](#) `c_init_cert_verify`

The `c_init_cert_verify` exchange occurs when the TLS client is being requested to authenticate during the TLS handshake. According to [\[RFC8446\]](#) client authentication during the TLS handshake is not valid with a PSK or PSK based authentication. As a result, ECHDE needs to be agreed. In addition, as `c_init_cert_verify` initiates a LURK session, the ECDHE shared secrets have been generated by the TLS client as opposed to the CS.

The CS completes and returns the signature as described in [Section 5.10](#). When further LURK exchanges are expected, the CS generates also the Finished message in order to be able to complete the latter requests for post authentication.

This exchange is followed by a `c_post_hand_auth` exchange.



```

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

```

```

struct{
    uint8 tag;
    select (tag.last_exchange){
        case False:
            uint32 session_id;
    }
    Ephemeral ephemeral;
    Signature signature;
}CInitCertVerifyResponse

```

freshness, session\_id are respectively defined in [Section 5.4](#) and [Section 5.3](#).

tag is defined in [Section 5.8](#). The TLS client that does not expect an additional post handshake authentication MUST set it tag.last\_message. A consequence is that a TLS client that does not enable post authentication MUST set the tag.last\_message.

The CS MUST set the tag.last\_message when the TLS client does not enabled post handshake authentication or when no further post handshake authentication is expected.

ephemeral is defined in [Section 5.5](#). The TLS client MUST set the ephemeral\_method to 'secret\_provided'.

handshake is defined in [Section 5.2](#) and must be set to ECDHE agreed. Note that PSK may be proposed if these PSKs are not provisioned in the CS. The handshake messages are also in clear.

#### [7.4.](#) c\_init\_client\_hello

The c\_init\_ephemeral\_binder exchange occurs when the TLS client needs to generate the ClientHello as well as the early secrets. In fact the generation of the ClientHello may require the CS to generate the (EC)DHE private key and returns the public part as well as the binder\_key to generate the binders. On the other hand, the generation of the early secrets requires the ClientHello to be completed. As a result, the CS will be expected to complete the ClientHello from a potential partial ClientHello. More specifically, when binders are needed, the partial Client Hello does not contain the OfferedPsks structure, that is the PreSharedKeyExtension. The latter structure is simply stripped from the ClientHello, without any further changes, such as changing the lengths for example. It is entirely built by the CS and appended as the last extension as described in [section 4.2.11](#) of The reason for having CPskID as opposed to the identity structure is that nothing prevents identities of two different NewSessionTickets to collide. CPskID are managed by the CS of the TLS client to prevent such collisions and are provided during the registration of the NewSessionTickets c\_register\_tickets (see [[RFC8446](#)]). Note that extension\_type as well as the 16 bit length of the OfferedPsks remain present. The PreSharedKeyExtension structure of the ClientHello is built from a list of CPskIDs where each CPskID designates a PSK with an identifier managed by the CS of the TLS client. The PSK can be associated to a NewSessionTicket in which case the CPskID will be used to designate the NewSessionTicket and its associated identity structure ([Section 7.7](#)). When the PSK is provisioned, the TLS client needs to be configured with it. When the CS is not able to generate the PreSharedKeyExtension an invalid\_identity error SHOULD be raised.

Note that when PSK is not proposed or when PSK are not registered in the CS, the ClientHello is fully provided - without the PreSharedKeyExtension or with a completed PreSharedKeyExtension extension. The CS is expected to be able to distinguish between the two by for example, comparing the length of the handshake provided in c\_init\_client\_hello and the length indicated in the ClientHello.

Note that (EC)DHE may be generated when ECDHE or PSK-ECDHE authentication is proposed by the TLS client, while early secrets and binder\_key can only be requested when PSK is proposed. When the TLS client requests the generation of a (EC)DHE private key, the KeyShareClientHello MAY contain a list of KeyShareEntry (defined in [section 4.2.8](#) When an error is found regarding the KeyShareClientHello, the CS SHOULD raise an invalid\_ephemeral error. Note that according to [\[RFC8446\]](#) ). When provided, these structures contains the group but are being stripped the key\_exchange\_value, while all other fields - including the lengths - are left unchanged. [Section 5.5](#) when the ephemeral\_method is set to 'no\_secret', the resulting list is empty.

This exchange is followed by a c\_client\_hello or a c\_hand\_and\_app\_secret.

```
struct{
    uint32 c_psk_id
}CPskID
```

```
struct{
    uint32 session_id
    Freshness freshness
    Ephemeral ephemeral ephemeral_method=secret_generate or no_secret
    Handshake handshake<0..2^32> //RFC8446 section 4
    CPskID c_psk_id_list<0..2^8-1>
    uint16 secret_request;
}CInitClientHello
```

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

session\_id, freshness and ephemeral, secret\_request and secret\_list are respectively defined in [Section 5.3](#), [Section 5.4](#), [Section 5.5](#), [Section 5.1](#) and [Section 5.9](#).

ephemeral is defined in [Section 5.5](#). With a single ClientHello or partial ClientHello, ephemeral\_method is set to secret\_generate when ECDHE and PSK-ECDHE authentication are being proposed or no\_secret when only PSK is proposed or when the ECDHE is generated

by the TLS client.

Note that even though the ClientHello requests multiple KeyShareEntries, a single ephemeral method is provided.

handshake is defined in [Section 5.2](#). When a partial ClientHello is provided, PSK or PSK-ECDHE MUST be enabled. When a ClientHello is provided, PSK or PSK-ECDHE may be proposed but with unprotected keys. ClientHello, HelloRetryRequest, ClientHello or ClientHello, HelloRetryRequest, partial ClientHello MAY be provided. However, there are cases this is not possible, typically when the HellRequestRetry does not contain a key\_share extension, the (EC)DHE shared secret is generated with the (EC)DHE generated associated to the first ClientHello. When the (EC)DHE private key has been generated by the CS, the TLS client MUST use a c\_early\_secret LURK exchange as defined in [Section 7.5](#) in order to ensure the CS is aware of the (EC)DHE shared secret to generate the further secrets.

c\_psk\_id designate an 32 bit identifier for a PSK. This identifier is provided and managed by the CS of the TLS client to avoid collision of different PSK provided by different TLS servers.

c\_psk\_id\_list designates the list of CPskIDs. The list is used by the CS to build the OfferedPsk structure - including the PSKIdentity structure. The list of identities in the OfferedPsk MUST be the same as the one of the c\_psk\_id\_list.

ephemeral\_list When contains the different values of (EC)DHE public parts - i.e. the KeyShareEntries when the ephemeral\_method is set to secret\_generated. When the ephemeral\_method is set to no\_secret, the list is an empty list by construction of the ephemeral structure (see [Section 5.5](#)).

## [7.5](#). c\_client\_hello

The c\_client\_hello exchange occurs after a TLS server responds to a ClientHello generated using a c\_init\_client\_hello defined in [Section 7.4](#) is being responded a HelloRetryRequest by the TLS server. While in some cases, re-initiating a LURK exchange with a \_init\_client\_hello MAY be considered, this document RECOMMENDS to

proceed as follows when a HelloRetryRequest is received:

- \* If the first ClientHello has been generated via a `c_init_client_hello`, use `c_client_hello` to generate the second ClientHello
- \* If the first ClientHello has not been generated via a `c_init_client_hello`, consider generating the second ClientHello via `c_init_client_hello`.

This exchange is followed by a `c_hand_and_app_secret`.

```
struct{
    uint32 session_id
    Freshness freshness
    Ephemeral ephemeral ephemeral_method=secret_generate or no_secret
    Handshake handshake<0.. $2^{32}$ > //RFC8446 section 4
    uint16 secret_request;
}CClientHello
```

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

`session_id`, `ephemeral`, `secret_request` and `secret_list` are respectively defined in [Section 5.3](#), [Section 5.5](#), [Section 5.1](#) and [Section 5.9](#).

`handshake` is defined in [Section 5.2](#). The handshake MUST contain a HelloRetryRequest and a ClientHello or partial ClientHello. The same restrictions as defined in [Section 7.4](#) apply to the ClientHello

#### [7.6](#). `c_hand_and_app_secret`

The `c_hand_and_app_secret` exchange occurs after a ServerHello is received and the TLS client request handshake secrets to decrypt (resp. encrypt) handshake messages sent by (resp. to) the server. Similarly the TLS client requests application secrets used to protect

the TLS session as well as other secrets such as exporter secrets.

Upon reception of the handshake the CS derives the handshake secrets and the `server_handshake_traffic_secret` as described in [\[RFC8446\] section 7.3](#) to decrypt the encrypted messages. The presence of a `CertificateRequest` indicates the TLS server expects the TLS client to authenticate via a `CertificateVerify` message. If the CS protects a private key associated to the TLS client, the CS MUST provide the necessary information to the TLS client. Otherwise, the `CertificateRequest` is ignored by the CS.

When the CS generates the signature, the presence of the certificate, the signature and `sig_algo` is indicated by setting `tag.cert_request`. Unlike on the TLS server, where the TLS server indicates the certificate to choose as well as the signature scheme to select, on the TLS client, such decision is left to the CS. The choice of the signature algorithm and certificate is performed by the CS as described in When `resumption_master_secret` is requested by the TLS client, or when further exchanges between the TLS client and the CS

are expected, the CS generates the `CertificateVerify` and `Finished` message to synchronize the TLS handshake context. The `Certificate`, respectively `CertificateVerify` and `Finished` message are generated as described in [\[RFC8446\] section 4.4.2.3](#), [\[RFC8446\] section 4.4.2](#), [section 4.4.3](#), and [section 4.4.4](#).

This exchange is followed by a `c_post_hand_auth`, `c_register_ticket` exchange.

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

struct{
    uint8 tag;
    uint32 session_id;
    Secret secret_list<0..2^16-1>;
    select( tag.cert_request){
```

```

    case true:
        LURKTLS13Certificate certificate;
        SignatureScheme sig_algo;
        Signature signature;
    }
}CHandAndAppSecretRequest

```

session\_id, secret\_request certificate signature, and secret\_list are respectively described in [Section 5.3](#), [Section 5.1](#), [Section 5.7](#), [Section 5.10](#) and [Section 5.9](#)

tag is defined in [Section 5.8](#) and indicates whether the further exchanges are expected or not. If the TLS client or the TLS server do not expect to perform session resumption or have not enabled post handshake authentication the tag.last\_message SHOULD be set.

ephemeral is defined in [Section 5.5](#). Since ClientHello as already been sent, the purpose of the ephemeral is to provide the (EC)DHE shared secret to perform the key schedule and ephemeral\_method MUST NOT be set to secret\_generated.

handshake is defined in [Section 5.2](#) and includes the ServerHello up to the server Finished. These messages are passed to the CS encrypted.

sig\_algo is defined in [Section 5.10](#) and defines the algorithm chosen by the CS.

## [7.7](#). c\_register\_tickets

The c\_register\_ticket is only used when the TLS client intend to perform session resumption. The LURK client MAY provide one or multiple NewSessionTickets. These tickets will be helpful for the session resumption to bind the PSK value to some identities. As the NewSessionTicket's identities may collide when being provided by multiple TLS servers, the 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 7.4](#)). When too many tickets are provided, the 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.
} RegisterTicketsRequest;

struct {
    uint8 tag
    uint32 session_id
    CPskID c_spk_id_list<0..2^8-1>
} RegisterTicketsResponse;

```

## 8. Security Considerations

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

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

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

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



Future session may be impacted if an attacker is able to authenticate a future session based on what it learns from a current session. ECDHE authentication relies on cryptographic signature and an ongoing TLS handshake. The robustness of the signature depends on the signature scheme and the unpredictability of the TLS Handshake. PSK authentication relies on not revealing the PSK. The CS does not reveal the PSK. TLS 1.3 has been designed so secrets generated do not disclose the PSK as a result, secrets provided by the Cryptographic do not reveal the PSK. NewSessionTicket reveals the identity (ticket) of a PSK. NewSessionTickets.ticket are expected to be public data. Its 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 attacker gets 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 the attacker. This corresponds to the case where the TLS client is corrupted.

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

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

## [9.](#) IANA Considerations

## [10.](#) Acknowledgments

## 11. References

### 11.1. 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>>.
- [RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", [RFC 7924](#), DOI 10.17487/RFC7924, July 2016, <<https://www.rfc-editor.org/info/rfc7924>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in [RFC 2119](#) Key Words", [BCP 14](#), [RFC 8174](#), DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", [RFC 8446](#), DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.

### 11.2. Informative References

- [I-D.mglt-lurk-lurk]  
Migault, D., "LURK Protocol version 1", Work in Progress, Internet-Draft, [draft-mglt-lurk-lurk-00](#), 9 February 2018, <<https://www.ietf.org/archive/id/draft-mglt-lurk-lurk-00.txt>>.
- [I-D.mglt-lurk-tls12]  
Migault, D. and I. Boureau, "LURK Extension version 1 for (D)TLS 1.2 Authentication", Work in Progress, Internet-Draft, [draft-mglt-lurk-tls12-04](#), 25 January 2021, <<https://www.ietf.org/internet-drafts/draft-mglt-lurk-tls12-04.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 its 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.

TLS client

TLS Server

```

Key   ^ ClientHello
Exch  | + key_share
      v + signature_algorithms ----->
                                     ServerHello ^ Key
                                     + key_share v Exch
                                     {EncryptedExtensions} Server Params
                                     {Certificate} ^
                                     {CertificateVerify} | Auth
                                     {Finished} v
      <----- [Application Data*]
{Finished} ----->
[Application Data] <-----> [Application Data]

```

The TLS server interacts with the 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 the 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 the CS

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

In this scenario, the CS is the only entity that knows the private ECDHE key and the shared secret, and only the CS is able to compute the secrets. The CS indicates the exchange is final by setting the tag to `last_message`, 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`

`ephemeral_method = secret_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 the CS as only the shared secret is necessary to generate the signature. This is indicated by the `ephemeral_method` set to `secret_provided`. No (EC)DHE values are returned by the CS as these have already been generated by the TLS server. However, the TLS server has all the necessary material to generate the secrets and the only information that the CS owns and that is not known to the TLS server is the

private key (associated to the certificate) used to generate the signature. This means that if 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, the CS is the only entity that knows the private ECDHE key. Only the CS is able to generate the signature. Both the CS and the TLS server are able to compute all secrets. The CS indicates the exchange is final by setting the tag to `last_message`, 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 appropriate keys.

TLS server

LURK client

Cryptographic Service

`SInitCertVerifyRequest`

`tag.last_exchange=True` ----->

`freshness = sha256`

`ephemeral`

`ephemeral_method = secret_provided`

`key`

`group = x25519`

`shared_secret = shared_secret`

`handshake = handshake`

`certificate = finger_print`

`secret_request = h_s, h_c, a_s, and a_c`

`sig_algo = ed25519`

`SInitCertVerifyResponse`

`tag.last_exchange=True`

`secret_list`

`signature = sig`

<-----

## [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, the 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, the CS sort of achieves a delegation to the TLS server.

Migault

Expires 27 January 2022

[Page 40]

---

Internet-Draft

LURK/TLS 1.3

July 2021

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

ECDHE authentication is performed with the 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 the 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.

TLS server

LURK client

Cryptographic Service

    SInitCertVerifyRequest

        tag.last\_exchange=False

        session\_id = session\_id\_tls\_server      ----->

        freshness = sha256

        ephemeral

            ephemeral\_method = secret\_generated

        handshake = handshake (x25519)

        certificate = finger\_print

        secret\_request = h\_s, h\_c, a\_s, a\_c

        sig\_algo = ed25519

    SInitCertVerifyResponse

```

tag.last_exchange=False
session_id = session_id_cs
ephemeral
  key
    group = x25519,
    key_exchange = public_key
secret_list
signature = sig
<-----

```

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 the CS by sending a SNewTicketRequest. The tag.last\_exchange set to False indicates to the 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 the 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 the CS and will be anyway ignored by the CS, so the secret\_request has its r bit unset.

As depicted below, the 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}.

TLS server

LURK client

SNewTicketRequest

tag.last\_exchange=False

session\_id = session\_id\_cs

Cryptographic Service

```

handshake = client Finished
certificate
  certificate_type = empty
ticket_nbr
secret_request  ----->
                                SNewTicketResponse
                                tag.last_exchange=False
                                session_id = session_id_tls_server
                                secret_list
                                <----- ticket_list

```

### [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:

TLS client		TLS Server
ClientHello		
+ key_share		
+ psk_key_exchange_mode		
+ pre_shared_key	----->	
		ServerHello
		+ pre_shared_key
		+ key_share
		{EncryptedExtensions}
		{Finished}
	<-----	[Application Data*]

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 the 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 the 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 the 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.

The 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`.

TLS server

LURK client

Cryptographic Service

```
SInitEarlySecretRequest ----->
    session_id = session_id_tls_server
```

```

freshness = sha256
selected_identity = 0
handshake = ClientHello
secret_request = b

                                SInitEarlySecretResponse
                                session_id = session_id_cs
<----- secret_list = 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 ephemeral\_method to secret\_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.

The 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. The 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
    SHandshakeAndAppRequest
        tag.last_exchange = False
        session_id = session_id_cs
        ephemeral
            ephemeral_method = secret_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 the 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.

```

TLS client                                TLS Server

Key   ^ ClientHello
Exch  | + key_share
      v + signature_algorithms ----->
                                ServerHello ^ Key
                                + key_share v Exch
                                {EncryptedExtensions} Server Params
                                {Certificate} ^
                                {CertificateVerify} | Auth
                                {Finished} v
                                <----- [Application Data*]
{Finished} ----->
[Application Data] <----- [Application Data]

```

If the TLS client generates the (EC)DHE private key, no interaction with the 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 the 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 identifier of the session that is being assigned by the TLS client for future inbound LURK message responses sent by the CS. Similarly, the CS advertises its `session_id_cs`. freshness is set to sha256, and the `ClientHello.random` is generated as described in [Section 5.4](#). 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.

The 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`.

TLS client

LURK client

Cryptographic Service

`CInitClientHello`

`session_id = session_id_tls_client`

`freshness = sha256`

`ephemeral`

`ephemeral_method = secret_generated`

`handshake = ClientHello(x25519, x488, ... )`

`c_psk_id_list = []`

`secret_request = []` ----->

`CInitClientHello`

`session_id=session_id_cs`

`ephemeral_list`

`key`

`group = x25519,`

```
        key_exchange = public_key
        ephemeral_method = secret_generated
        key
        group = x488,
        key_exchange = public_key
    secret_list=[]
```

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_message` is unset. The `session_id` takes the value advertised by each party during the previous `c_init_client_hello` exchange. Since the 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, `ephemeral_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.

The CS generates the handshake secrets and the associated key to decrypt the encrypted messages. As no CertificateRequest has been found, the CS does not compute the signature that would authenticate the TLS client. In this section, we assume the 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, the CS computes the Finished message to generate the `resumption_master_secret`.

The CS returns the response by unsetting the `tag.last_message` 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\_message=False

session\_id=session\_id\_cs

ephemeral

ephemeral\_method = no\_secret

handshake = ServerHello, {EncryptedExtensions}...,{Finished}.

secret\_request = h\_c, h\_s, a\_c, a\_s -----&gt;

CHandAndAppSecretResponse

tag

last\_message=False

cert\_request=False

session\_id=session\_id\_tls\_clt

ephemeral\_list = []

secret\_request = h\_s, h\_c, a\_s, and a\_c

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 the 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\_message. For each registered NewSessionTicket, the CS returns c\_spk\_id that will use for further references. The c\_spk\_ids are managed by the 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.

```

TLS client
LURK client                                Cryptographic Service
RegisterTicketsRequest
  tag.last_message=False
  session_id=session_id_cs
  ticket_list = [NewSessionTicket]
  ----->
                                RegisterTicketsResponse
                                last_message=False
                                session_id=session_id_tls_clt
<----- c_spk_id_list = [nst_id]
```

#### [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





```
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 the CS as described below:

TLS client

LURK client

Cryptographic Service

`CHandAndAppSecretRequest`

`tag.last_message=False`

`session_id=session_id_cs`

`ephemeral`

`ephemeral_method = secret_provided`

`shared_secret`

`handshake = ServerHello, {EncryptedExtensions}...,{Finished}.`

`secret_request = h_c, h_s, a_c, a_s ----->`

`CHandAndAppSecretResponse`

`tag`

`last_message=False`

`cert_request=False`

`session_id=session_id_tls_clt`

`ephemeral_list = []`

`secret_request = h_s, h_c, a_s, and a_c`

Upon receiving the response, the TLS client proceeds similarly to the TLS client described in [Appendix A.4](#).

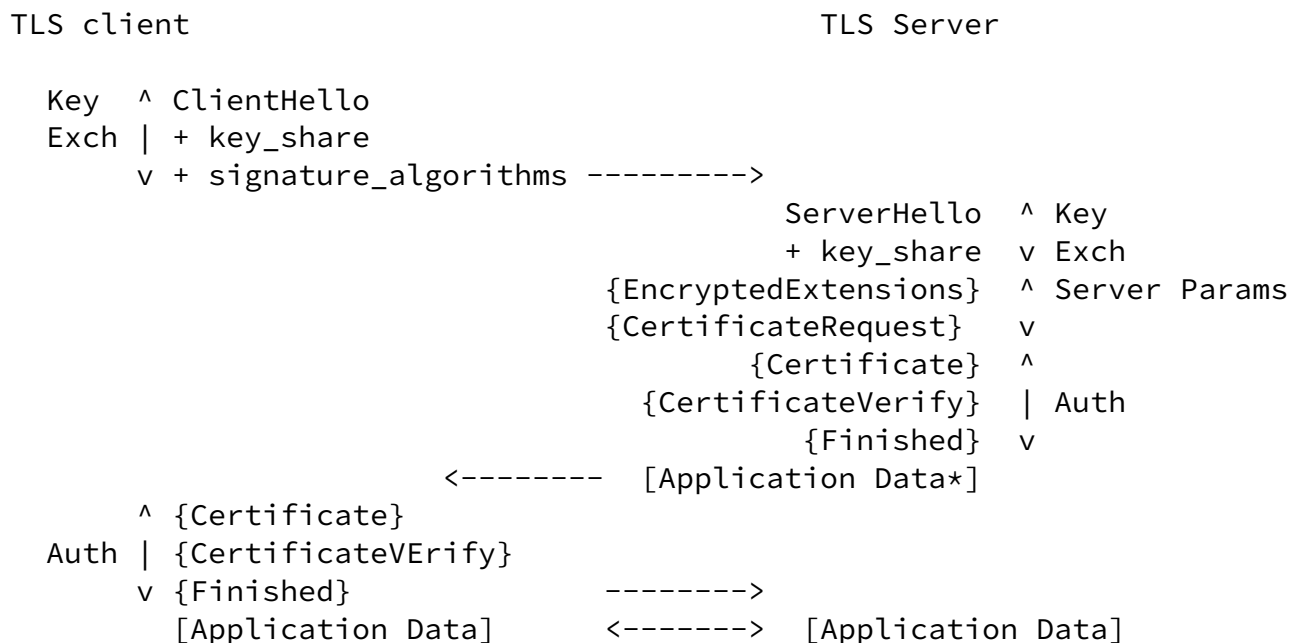
## [A.6](#). TLS client authenticated ECDHE

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 the 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 CertificateRequest 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](#).

The CS proceeds as described in [Appendix A.5](#) or [Appendix A.4](#). However, after the messages have been decrypted, the CS proceeds to the generation of the signature and returns the necessary information to build the CertificateVerify. The 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\_message=False

session\_id=session\_id\_cs

ephemeral

ephemeral\_method = secret\_provided

shared\_secret

handshake = ServerHello, {EncryptedExtensions}..., {Finished}.

secret\_request = h\_c, h\_s, a\_c, a\_s ----->

CHandAndAppSecretResponse

tag

last\_message=False

cert\_request=True

session\_id=session\_id\_tls\_clt

ephemeral\_list = []

secret\_request = h\_s, h\_c, a\_s, and a\_c

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.

Internet-Draft

LURK/TLS 1.3

July 2021

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 the 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 ephemeral\_method is necessarily set to secret\_provided. The handshake is set to the ClientHello ... server Finished, and the certificate carries the reference to the TLS client certificate, so the CS picks the appropriated private key. sig\_algo designates the signature algorithm.

TLS server

LURK client

Cryptographic Service

CInitCertVerifyRequest

tag.last\_exchange=True -----&gt;

freshness = sha256

ephemeral

ephemeral\_method = secret\_provided

key

group = x25519

shared\_secret = shared\_secret

handshake = hanshake

certificate

certificate\_type = finger\_print

sig\_algo = ed25519

CInitCertVerifyResponse

tag.last\_exchange=True

signature = sig

&lt;-----

#### [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 the 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_message` is set in order to accept future post handshake authentication request. `ephemeral_method` is set to `secret_provide` as the 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.

TLS server

LURK client

Cryptographic Service

`CInitPostHandAuthRequest`

`tag.last_message = False`

`session_id = session_id_tls_client`

`freshness = sha256`

`ephemeral`

`ephemeral_method = secret_provided`

`handshake = ClientHello ... client Finished CertificateRequest`

`certificate`

`certificate_type = finger_print`

`sig_algo ----->`

`CInitPostHandAuthResponse`

`tag.last_message = False`

```

                                session_id = session_id_cs
<----- signature = sig

```

#### [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 Appendix A.7.1 except that the LURK session does not need to be initiated, the shared secret is already known to the CS and the handshake is only constituted of the remaining CertificateRequest message.

Migault

Expires 27 January 2022

[Page 54]

Internet-Draft

LURK/TLS 1.3

July 2021

TLS server

LURK client

Cryptographic Service

CInitPostHandAuthRequest

tag.last\_message = False

session\_id = session\_id\_tls\_client

handshake = CertificateRequest

certificate

certificate\_type = finger\_print

sig\_algo ----->

CInitPostHandAuthResponse

tag.last\_message = False

session\_id = session\_id\_cs

<----- signature = sig

Author's Address

Daniel Migault

Ericsson

8275 Trans Canada Route

Saint Laurent, QC 4S 0B6

Canada

Email: daniel.migault@ericsson.com

