Workgroup: tls
Internet-Draft:
draft-celi-wiggers-tls-authkem-00
Published: 12 July 2021
Intended Status: Informational
Expires: 13 January 2022
Authors: S. Celi
Cloudflare
P. Schwabe
Radboud University & MPI S&P
D. Stebila
University of Waterloo
N. Sullivan
Cloudflare
T. Wiggers
Radboud University

KEM-based Authentication for TLS 1.3

Abstract

This document gives a construction for KEM-based authentication in TLS 1.3. The overall design approach is a simple: usage of Key Encapsulation Mechanisms (KEM) for certificate-based authentication.

Discussion Venues

This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the mailing list (), which is archived at .

Source for this draft and an issue tracker can be found at https://github.com/claucece/draft-celi-wiggers-tls-authkem.

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 13 January 2022.

Table of Contents

1. Introduction

DISCLAIMER: This is a work-in-progress draft.

This document gives a construction for KEM-based authentication in TLS 1.3. The overall design approach is a simple: usage of Key Encapsulation Mechanisms (KEM) for certificate-based authentication. Authentication happens via asymmetric cryptography by the usage of KEMs advertised as the long-term KEM public keys in the Certificate.

TLS 1.3 is in essence a signed key exchange protocol (if using certificate-based authentication). Authentication in TLS 1.3 is achieved by signing the handshake transcript. KEM-based authentication provides authentication by deriving a shared secret that is encapsulated against the public key contained in the certificate. Only the holder of the private key corresponding to the certificate's public key can derive the same shared secret and thus decrypt it's peers messages.

This approach is appropriate for endpoints that have KEM public keys. Though this is currently rare, certificates could be issued with (EC)DH public keys as specified for instance in [RFC8410], or using a delegation mechanism, such as delegated credentials [I-D.ietf-tls-subcerts].

In this proposal we use the DH-based KEMs from [I-D.irtf-cfrg-hpke]. We believe KEMs are especially worth discussing in the context of the TLS protocol because NIST is in the process of standardizing post-quantum KEM algorithms to replace "classic" key exchange (based on elliptic curve or finite-field Diffie-Hellman [NISTPQC]).

This proposal draws inspiration from [I-D.ietf-tls-semistatic-dh], which is in turn based on the OPTLS proposal for TLS 1.3 [KW16]. However, these proposals require a non-interactive key exchange: they combine the client's public key with the server's long-term key. This imposes a requirement that the ephemeral and static keys use the same algorithm, which this proposal does not require. Additionally, there are no post-quantum proposals for a non-interactive key exchange currently considered for standardization, while several KEMs are on the way.

2. Requirements Notation

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.

3. Terminology

The following terms are used as they are in [RFC8446]

client: The endpoint initiating the TLS connection.

connection: A transport-layer connection between two endpoints.

endpoint: Either the client or server of the connection.

handshake: An initial negotiation between client and server that establishes the parameters of their subsequent interactions within TLS.

peer: An endpoint. When discussing a particular endpoint, "peer" refers to the endpoint that is not the primary subject of discussion.

receiver: An endpoint that is receiving records.

sender: An endpoint that is transmitting records.

server: The endpoint that this did initiate the TLS connection. i.e. the peer of the client.

3.1. Key Encapsulation Mechanisms

As this proposal relies heavily on KEMs, which are not originally used by TLS, we will provide a brief overview of this primitive.

A Key Encapsulation Mechanism (KEM), defined as in [I-D.irtf-cfrg-hpke] as an internal API, is a cryptographic primitive that defines the methods Encap and Decap:

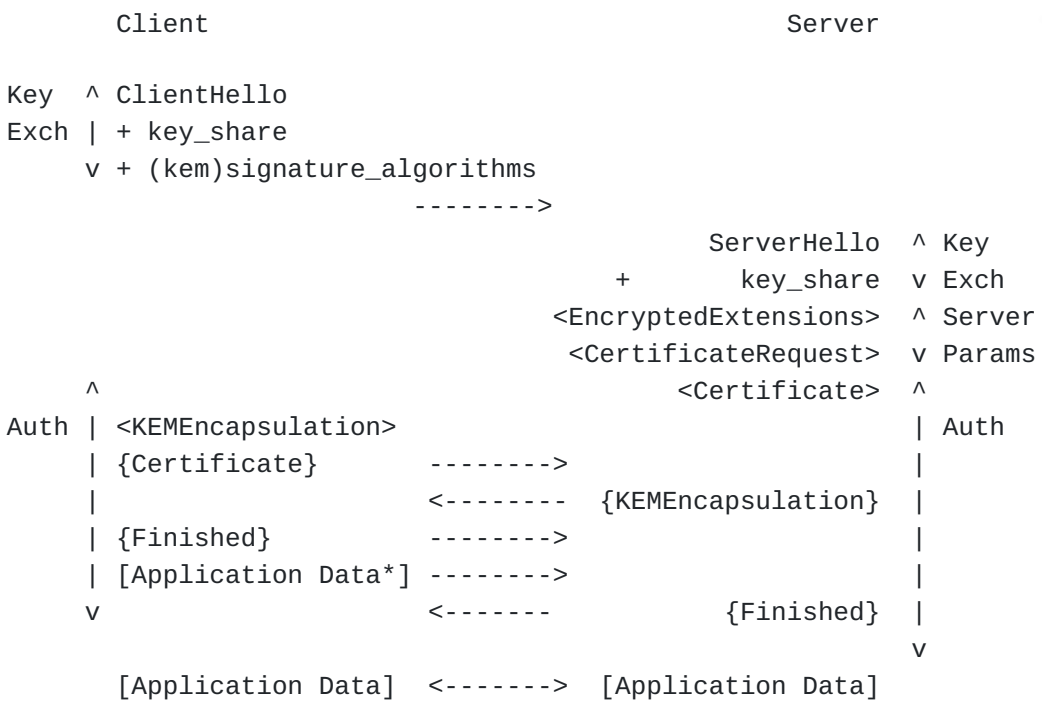
Encap(pkR): Takes a public key, and produces a shared secret and encapsulation.

Decap(enc, skR): Takes the encapsulation and the private key. Returns the shared secret.

Note that we are using the internal API for KEMs as defined in [I-D.irtf-cfrg-hpke].

4. Protocol Overview

Figure 1 below shows the basic full KEM-authentication handshake:



+ Indicates noteworthy extensions sent in the previously noted message.
* Indicates optional or situation-dependent messages/extensions that are not always sent.
<> Indicates messages protected using keys derived from a [sender]_handshake_traffic_secret.
{} Indicates messages protected using keys derived from a [sender]_authenticated_handshake_traffic_secret.
[] Indicates messages protected using keys derived from [sender]_application_traffic_secret_N.

Figure 1: Message Flow for KEM-Authentication Handshake

When using KEMs for authentication, the handshake can be thought of in four phases compared to the three ones from TLS 1.3. It achieves both confidentiality and authentication (certificate-based).

After the Key Exchange and Server Parameters phase of TLS 1.3 handshake, the client and server exchange implicitly authenticated messages. KEM-based authentication uses the same set of messages every time that certificate-based authentication is needed. Specifically:

*Certificate: The certificate of the endpoint and any per-certificate extensions. This message is omitted by the client if the server did not send CertificateRequest (thus indicating that the client should not authenticate with a certificate). The Certificate MUST include the long-term KEM public key.

* KEMEncapsulation: A key encapsulation against the certificate's long-term public key, which yields an implicitly authenticated shared secret.

Upon receiving the server's messages, the client responds with its Authentication messages, namely Certificate and KEMEncapsulation (if requested). If client authentication was not requested, the Client sends its Finished message.

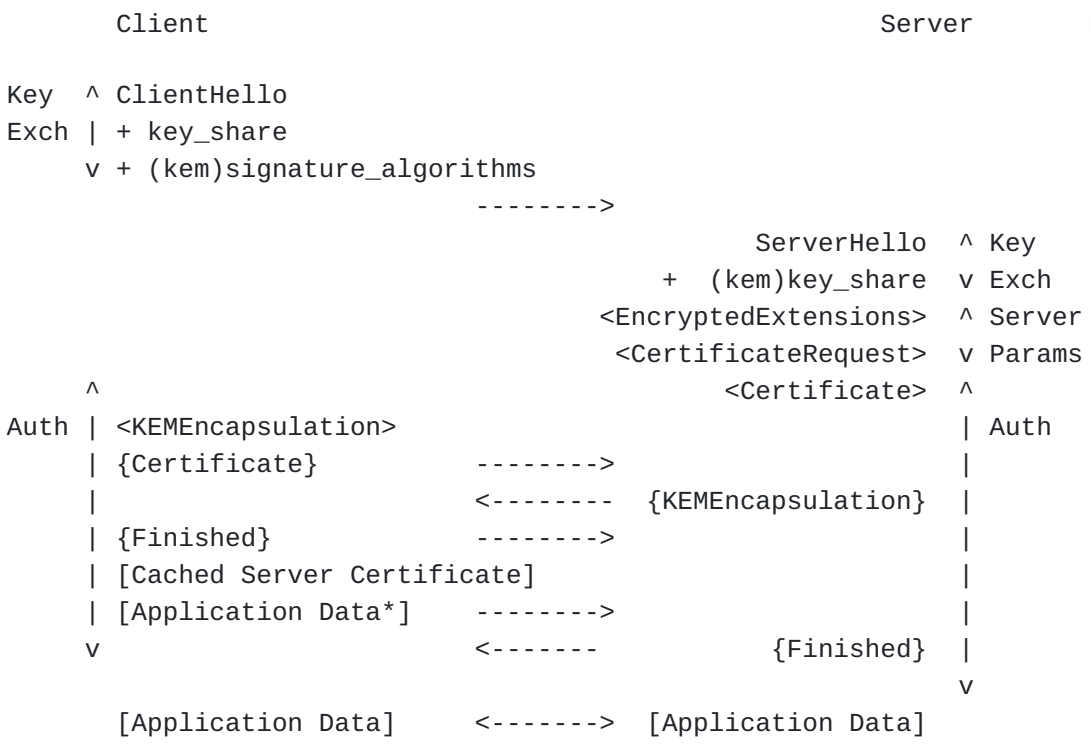
Upon receiving the client's messages, the server responds with its Finished message, which achieves explicit authentication. Upon receiving the server's Finished message, the client achieves explicit authentication.

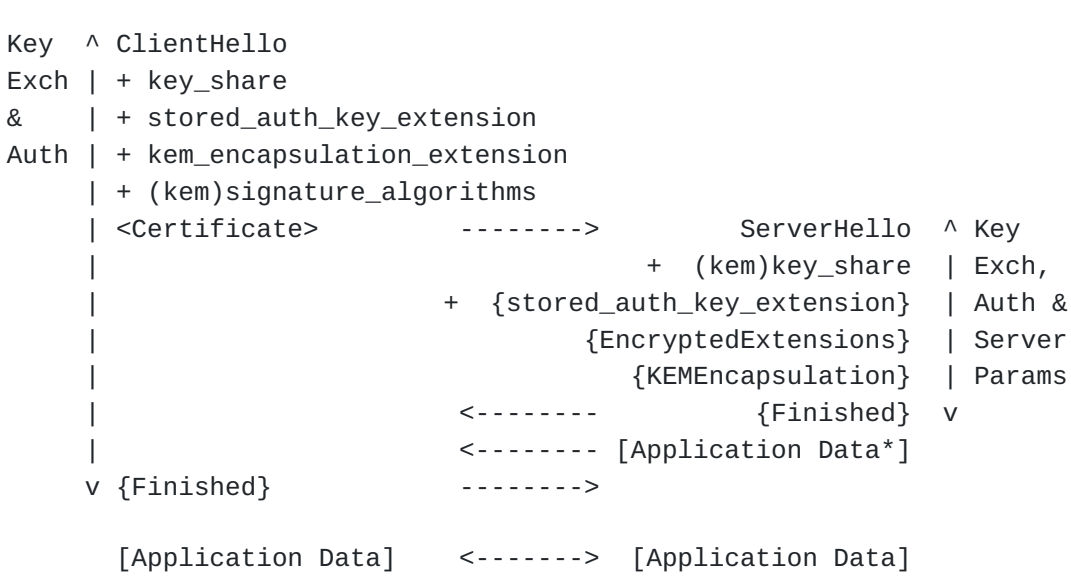
Application Data MUST NOT be sent prior to sending the Finished message, except as specified in Section 2.3 of [RFC8446]. Note that while the client may send Application Data prior to receiving the server's last explicit Authentication message, any data sent at that point is, of course, being sent to an implicitly authenticated peer. It is worth noting that Application Data sent prior to receiving the server's last Authentication message can be subject to a client downgrade attack. Full downgrade resilience is only achieved when explicit authentication is achieved: when the Client receives the Finished message from the Server.

4.1. Prior-knowledge KEM-Auth

Given the added number of round-trips of KEM-based auth compared to the TLS 1.3, the handshake can be improved by the usage of pre-distributed KEM authentication keys to achieve explicit authentication and full downgrade resilience as early as possible. A peer's long-term KEM authentication key can be cached in advance, as well.

Figure 2 below shows a pair of handshakes in which the first handshake establishes cached information and the second handshake uses it:





In some applications, such as in a VPN, the client already knows that the server will require mutual authentication. This means that a client can proactively authenticate by sending its certificate as early in the handshake as possible. The client's certificate has to be sent encrypted by using the shared secret derived from the kem_encapsulation message.

5. Negotiation

Clients will indicate support for this mode by negotiating it as if it were a signature scheme (part of the SignatureScheme extension). We thus add these new signature scheme values (even though, they are not signature schemes) for the KEMs defined in [I-D.irtf-cfrg-hpke] Section 7.1. Note that we will be only using their internal KEM's API defined there.

```
enum {
  dhkem_p256_sha256   => TBD,
  dhkem_p384_sha384   => TBD,
  dhkem_p521_sha512   => TBD,
  dhkem_x25519_sha256 => TBD,
  dhkem_x448_sha512   => TBD,
}
```

When present in the signature_algorithms extension, these values indicate KEM-Auth with the specified key exchange mode. These values MUST NOT appear in signature_algorithms_cert.

In order to be used for KEM-Auth, this algorithms have to be added to the SignatureScheme extension sent in the ClientHello message, and supported by the server.

6. Handshake protocol

The handshake protocol is used to negotiate the security parameters of a connection, as in TLS 1.3. It uses the same messages, expect for the addition of a KEMEncapsulation message and does not use the CertificateVerify one.

```
enum {
    ...
    encrypted_extensions(8),
    certificate(11),
    kem_encapsulation(tbd),
    certificate_request(13),
    ...
    message_hash(254),
    (255)
  } HandshakeType;

struct {
    HandshakeType msg_type;    /* handshake type */
    uint24 length;             /* remaining bytes in message */
    select (Handshake.msg_type) {
        ...
        case encrypted_extensions:  EncryptedExtensions;
        case certificate_request:   CertificateRequest;
        case certificate:           Certificate;
        case kem_encapsulation:     KEMEncapsulation;
        ...
    };
} Handshake;
```

Protocol messages MUST be sent in the order defined in Section 4. A peer which receives a handshake message in an unexpected order MUST abort the handshake with an "unexpected_message" alert.

6.1. Key Exchange Messages

KEM-Auth uses the same key exchange messages as TLS 1.3 with this exceptions:

- Two extensions can be added to the ClientHello message: "stored_auth_key" and "kem_encapsulation".

- One extensions can be added to the ServerHello message: "stored_auth_key".

KEM-Auth preserves the same cryptographic negotiation with the addition of the KEM algorithms to the signature_algorithms.

6.1.1. Client Hello

KEM-Auth uses the ClientHello message as described for TLS 1.3. When used in a pre-distributed mode, however, two extensions are mandatory: "stored_auth_key" and "kem_encapsulation" for server authentication. This extensions are described later in the document.

Note that in KEM-Auth with pre-distributed information, the client's Certificate message gets send alongside the ClientHello one for mutual authentication.

6.1.2. Server Hello

KEM-Auth uses the ServerHello message as described for TLS 1.3. When used in a pre-distributed mode, however, one extension is mandatory: "stored_auth_key" for server authentication. This extension is described later in the document.

When the ServerHello message is received:

- the client and server derive handshake traffic secrets client_handshake_traffic_secret and server_handshake_traffic_secret which are used to encrypt subsequent flows in the handshake

- the "handshake secret" is derived: dHS which is kept as the current secret state of the key schedule.

Refer to Section 8.1 for information on how to derive this secrets.

6.1.3. Hello Retry Request

KEM-Auth uses the ServerHello message as described for TLS 1.3. When used in a pre-distributed mode for mutual authentication, a HelloRetryRequest message can be sent, but the client's Certificate message is ignored.

6.1.4. Extensions

A number of KEM-Auth messages contain tag-length-value encoded extensions structures. We are adding those extensions to the ExtensionType list from TLS 1.3.

```
enum {
    ...
    signature_algorithms_cert(50),              /* RFC 8446 */
    key_share(51),                              /* RFC 8446 */
    kem_encapsulation (TBD),                    /* RFC TBD */
    stored_auth_key(TBD),                       /* RFC TBD */
    (65535)
} ExtensionType;
```

The table below indicates the messages where a given extension may appear:

```
+--------------------------------------+-------------+
| Extension                            |    KEM-Auth |
+--------------------------------------+-------------+
| stored_auth_key [RFCTBD]             |      CH, SH |
|                                      |             |
| kem_encapsulation  [RFCTBD]          |          CH |
|                                      |             |
+--------------------------------------+-------------+
```

6.1.4.1. Stored Auth Key

This document defines a new extension type ("stored_auth_key(TBD)"), which is used in ClientHello and ServerHello messages. The extension type is specified as follows.

```
struct {
     stored_auth_key(TBD), (65535)
} ExtensionType;
```

The extension_data field of this extension, when included in the ClientHello, MUST contain the StoredInformation structure. The client MAY send multiple StoredObjects of the same StoredInformationType. This may, for example, be the case when the client has cached multiple public keys from the server.

```
enum {
     uint8 pub_key = 0;
} StoredInformationType;

struct {
     select (type) {
       case client:
         StoredInformationType type;
         opaque hash_value<1..255>;
       case server:
         StoredInformationType type;
     } body;
} StoredObject;

struct {
     StoredObject stored_auth_key<1..2^16-1>;
} StoredInformation;
```

This document defines the following type:

* 'pub_key' type for not sending the complete server certificate message: With the type field set to 'pub_key', the client MUST include the fingerprint of the Public Key of the end-entity certificate in the hash_value field. For this type, the fingerprint MUST be calculated using the procedure below, using

the Public Key (represented using the Subject Public Key Info representation, as defined in [RFC5869], Section 4.1.2.7) as the input data.

The fingerprint calculation proceeds this way:

1. Compute the SHA-256 hash of the input data. Note that the computed hash only covers the input data structure (and not any type and length information of the record layer).

2. Use the output of the SHA-256 hash.

The purpose of the fingerprint provided by the client is to help the server select the correct information. The fingerprint identifies the server public key (and the corresponding private key) for use with the rest of the handshake.

If this extension is not present, the kem_encapsulation extension MUST not be present as well. If present, it will be ignored.

6.1.5. Implicit Authentication Messages

As discussed, KEM-Auth generally uses a common set of messages for implicit authentication and key confirmation: Certificate and KEMEncapsulation. The CertificateVerify message MUST NOT be sent.

The computations for the Authentication messages take the following inputs:

- The certificate and authentication key to be used.

- A Handshake Context consisting of the set of messages to be included in the transcript hash.

- A Shared Secret Key (from the key exchange operations) to be used to compute an authenticated handshake shared key.

- A Handshake Context consisting of the set of messages to be included in the transcript hash.

Based on these inputs, the messages then contain:

Certificate: The certificate to be used for authentication, and any supporting certificates in the chain.

KEMEncapsulation: The KEM encapsulation against the end-entity certificate's public key(s).

KEM-Auth follows the TLS 1.3 key schedule, which applies a sequence of HKDF operations to the Handshake Secret Keys and the handshake context to derive:

- the client and server authenticated handshake traffic secrets client_handshake_authenticated_traffic_secret and server_handshake_authenticated_traffic_secret which are used to encrypt subsequent flows in the handshake

- updated secret state dAHS of the key schedule.

- a Master Key.

6.1.6. Certificate

KEM-Auth uses the same Certificate message as TLS 1.3.

The end-entity Certificate or the RawPublicKey MUST contain or be a KEM public key.

Note that we are only specifying here the algorithms in the end-entity Certificate. All certificates provided by the server or client MUST be signed by an authentication algorithm advertised by the server or client.

6.1.7. KEM Encapsulation

This message is used to provide implicit proof that an endpoint possesses the private key(s) corresponding to its certificate by sending the appropriate parameters that will be used to calculate the implicitly authenticated shared secret.

The calculation of the shared secret also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message has been received and prior to the Finished message.

Structure of this message:

```
struct {
    opaque encapsulation<0..2^16-1>;
} KEMEncapsulation;
```

The encapsulation field is the result of a Encaps(pkR) function. The Encapsulation() function will also result on a shared secret (ssS or ssC, depending on the Server or Client executing it respectively) which is used to derive the AHS or MS secrets.

If the KEMEncapsulation message is sent by a server, the authentication algorithm MUST be one offered in the client's signature_algorithms extension unless no valid certificate chain can be produced without unsupported algorithms.

If sent by a client, the authentication algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the signature_algorithms extension in the CertificateRequest message.

In addition, the authentication algorithm MUST be compatible with the key(s) in the sender's end-entity certificate.

The receiver of a KEMEncapsulation message MUST perform the Decap(enc, skR) operation by using the sent encapsulation and the private key of the public key advertised in the end-entity certificate sent. The Decap(enc, skR) function will also result on a shared secret (ssS or ssC, depending on the Server or Client executing it respectively) which is used to derive the AHS or MS secrets.

6.1.8. Explicit Authentication Messages

As discussed, KEM-Auth generally uses a message for explicit authentication: Finished message. Note that in the non pre-distributed mode, KEM-Auth achieves explicit authentication only when the server sends the final Finished message (the client is only implicitly authenticated when they send their Finished message). In a pre-distributed mode, the server achieves explicit authentication when sending their Finished message (one round-trip earlier) and the client, in turn, when they send their Finished message (one round-trip earlier). Full downgrade resilience and forward secrecy is achieved once the KEM-Auth handshake completes.

The key used to compute the Finished message is computed from the Master Key using HKDF. Specifically:

```
server/client_finished_key =
  HKDF-Expand-Label(MasterKey,
                    server/client_label,
                    "", Hash.length)
server_label = "tls13 server finished"
client_label = "tls13 client finished"
```

Structure of this message:

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

The verify_data value is computed as follows:

```
server/client_verify_data =
      HMAC(server/client_finished_key,
           Transcript-Hash(Handshake Context,
                           Certificate*,
                           KEMEncapsulation*,
                           Finished**)
```

*Only included if present. ** The party who last sends the finished message in terms of flights includes the other party's Finished message.

Any records following a Finished message MUST be encrypted under the appropriate application traffic key as described in TLS 1.3. In particular, this includes any alerts sent by the server in response to client Certificate and KEMEncapsulation messages.

7. Cryptographic Computations

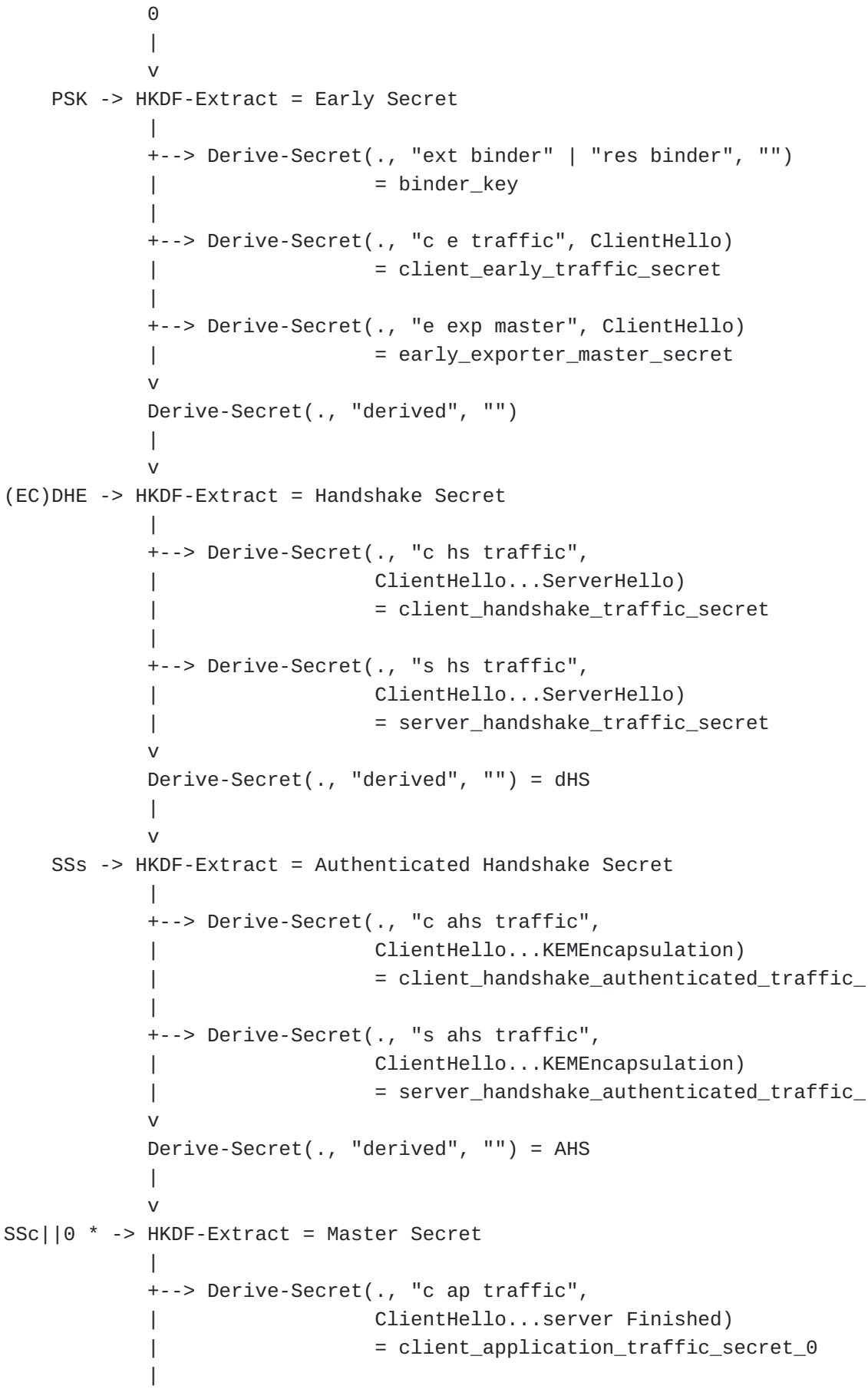
The KEM-Auth handshake establishes three input secrets which are combined to create the actual working keying material, as detailed below. The key derivation process incorporates both the input secrets and the handshake transcript. Note that because the handshake transcript includes the random values from the Hello messages, any given handshake will have different traffic secrets, even if the same input secrets are used.

7.1. Key schedule

KEM-Auth uses the same HKDF-Extract and HKDF-Expand functions as defined by TLS 1.3, in turn defined by [RFC5869].

Keys are derived from two input secrets using the HKDF-Extract and Derive-Secret functions. The general pattern for adding a new secret is to use HKDF-Extract with the Salt being the current secret state and the Input Keying Material (IKM) being the new secret to be added.

The key schedule proceeds as follows:


0
PSK -> HKDF-Extract = Early Secret
Derive-Secret(., "ext binder" | "res binder", "")
= binder_key
Derive-Secret(., "c e traffic", ClientHello)
= client_early_traffic_secret
Derive-Secret(., "e exp master", ClientHello)
= early_exporter_master_secret
Derive-Secret(., "derived", "")
(EC)DHE -> HKDF-Extract = Handshake Secret
Derive-Secret(., "c hs traffic",
ClientHello...ServerHello)
= client_handshake_traffic_secret
Derive-Secret(., "s hs traffic",
ClientHello...ServerHello)
= server_handshake_traffic_secret
Derive-Secret(., "derived", "") = dHS
SSs -> HKDF-Extract = Authenticated Handshake Secret
Derive-Secret(., "c ahs traffic",
ClientHello...KEMEncapsulation)
= client_handshake_authenticated_traffic_
Derive-Secret(., "s ahs traffic",
ClientHello...KEMEncapsulation)
= server_handshake_authenticated_traffic_
Derive-Secret(., "derived", "") = AHS
SSc||0 * -> HKDF-Extract = Master Secret
Derive-Secret(., "c ap traffic",
ClientHello...server Finished)
= client_application_traffic_secret_0

```
              +--> Derive-Secret(., "s ap traffic",
              |                  ClientHello...server Finished)
              |                  = server_application_traffic_secret_0
              |
              +--> Derive-Secret(., "exp master",
              |                  ClientHello...server Finished)
              |                  = exporter_master_secret
              |
              +--> Derive-Secret(., "res master",
                                 ClientHello...client Finished)
                                 = resumption_master_secret
```

The * means that if client authentication was requested the `SSc` value be used. Otherwise, the `0` value is used.

The operations to compute SSs or SSc from the client are:

```
SSs, encapsulation <- Encap(public_key_server)
               SSc <- Decap(encapsulation, private_key_client)
```

The operations to compute SSs or SSc from the server are:

```
               SSs <- Decap(encapsulation, priavte_key_server)
SSc, encapsulation <- Encap(public_key_client)
```

8. Security Considerations

*The academic works proposing KEM-Auth contain a in-depth technical discussion of and a proof of the security of the handshake protocol without client authentication [KEMTLS]. The work proposing the variant protocol [KEMTLSPDK] with pre-distributed public keys has a proof for both unilaterally and mutually authenticated handshakes.

*Application Data sent prior to receiving the server's last explicit authentication message (the Finished message) can be subject to a client downgrade attack, and has weaker forward secrecy compared to TLS 1.3. Full downgrade resilience and forward secrecy is achieved once the handshake completes.

*The client's certificate is kept secret from active observers by the derivation of the client_authenticated_handshake_secret, which ensures that only the intended server can read the client's identity.

*When the client opportunistically sends its certificate, it is not encrypted under a forward-secure key. This has similar considerations and trade-offs as 0-RTT data. If it is a replayed message, there are no expected consequences for security as the malicious replayer will not be able to decapsulate the shared secret.

*A client that opportunistically sends its certificate, SHOULD send it encrypted with a ciphertext that it knows the server will accept. Otherwise, it will fail.

8.1. Implicit authentication

Because preserving a 1/1.5RTT handshake in KEM-Auth requires the client to send its request in the same flight when the ServerHello message is received, it can not yet have explicitly authenticated the server. However, through the inclusion of the key encapsulated to the server's long-term secret, only an authentic server should be able to decrypt these messages.

However, the client can not have received confirmation that the server's choices for symmetric encryption, as specified in the ServerHello message, were authentic. These are not authenticated until the Finished message from the server arrived. This may allow an adversary to downgrade the symmetric algorithms, but only to what the client is willing to accept. If such an attack occurs, the handshake will also never successfully complete and no data can be sent back.

If the client trusts the symmetric algorithms advertised in its ClientHello message, this should not be a concern. A client MUST NOT accept any cryptographic parameters it does not include in its own ClientHello message.

If client authentication is used, explicit authentication is reached before any application data, on either client or server side, is transmitted.

9. References

9.1. Normative References

[I-D.ietf-tls-semistatic-dh] Rescorla, E., Sullivan, N., and C. A. Wood, "Semi-Static Diffie-Hellman Key Establishment for TLS 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-semistatic-dh-01, 7 March 2020, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-semistatic-dh-01>.

[I-D.ietf-tls-subcerts] Barnes, R., Iyengar, S., Sullivan, N., and E. Rescorla, "Delegated Credentials for TLS", Work in Progress, Internet-Draft, draft-ietf-tls-subcerts-10, 24 January 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-subcerts-10>.

[I-D.irtf-cfrg-hpke] Barnes, R. L., Bhargavan, K., Lipp, B., and C. A. Wood, "Hybrid Public Key Encryption", Work in Progress, Internet-Draft, draft-irtf-cfrg-hpke-10, 7 July 2021, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hpke-10>.

[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/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8410] Josefsson, S. and J. Schaad, "Algorithm Identifiers for Ed25519, Ed448, X25519, and X448 for Use in the Internet X.509 Public Key Infrastructure", RFC 8410, DOI 10.17487/RFC8410, August 2018, <https://www.rfc-editor.org/rfc/rfc8410>.

[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/rfc/rfc8446>.

9.2. Informative References

[KEMTLS] Stebila, D., Schwabe, P., and T. Wiggers, "Post-Quantum TLS without Handshake Signatures", DOI 10.1145/3372297.3423350, IACR ePrint https://ia.cr/2020/534, November 2020, <https://doi.org/10.1145/3372297.3423350>.

[KEMTLSPDK] Stebila, D., Schwabe, P., and T. Wiggers, "More Efficient KEMTLS with Pre-Shared Keys", May 2021.

[KW16] Krawczyk, H. and H. Wee, "The OPTLS Protocol and TLS 1.3", Proceedings of Euro S"P 2016 , 2016, <https://eprint.iacr.org/2015/978>.

[NISTPQC] NIST, ., "Post-Quantum Cryptography Standardization", 2020.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/rfc/rfc5280>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/rfc/rfc5869>.

Appendix A. Acknowledgements

This work has been supported by the European Research Council through Starting Grant No. 805031 (EPOQUE).

Authors' Addresses

Sofía Celi
Cloudflare

Email: cherenkov@riseup.net

Peter Schwabe
Radboud University & MPI S&P

Email: peter@cryptojedi.org

Douglas Stebila
University of Waterloo

Email: dstebila@uwaterloo.ca

Nick Sullivan
Cloudflare

Email: nick@cloudflare.com

Thom Wiggers
Radboud University

Email: thom@thomwiggers.nl