Workgroup: TLS Working Group
Internet-Draft:
draft-celi-wiggers-tls-authkem-01
Published: 7 March 2022
Intended Status: Informational
Expires: 8 September 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 a Key Encapsulation Mechanism (KEM)-based authentication mechanism in TLS 1.3. This proposal authenticates peers via a key exchange protocol, using their long-term (KEM) public keys.

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 8 September 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. 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 with digital signatures algorithms. 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 can 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 [RFC9180]. 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 an extra requirement: the ephemeral and static keys MUST 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.

1.1. Organization

After a brief introduction to KEMs, we will introduce the AuthKEM authentication mechanism. For clarity, we discuss unilateral and mutual authentication separately. Next, we introduce the abbreviated

AuthKEM handshake, and its opportunistic client authentication mechanism. In the remainder of the draft, we will discuss the necessary implementation mechanics, such as code points, extensions, new protocol messages and the new key schedule.

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 responded to the initiation of 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. Other cryptographic operations will be discussed later.

A Key Encapsulation Mechanism (KEM) is a cryptographic primitive that defines the methods Encapsulate and Decapsulate. In this draft, we extend these operations with context separation strings:

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

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

We implement these methods through the KEMs defined in [RFC9180] to export shared secrets appropriate for using with the HKDF in TLS 1.3:

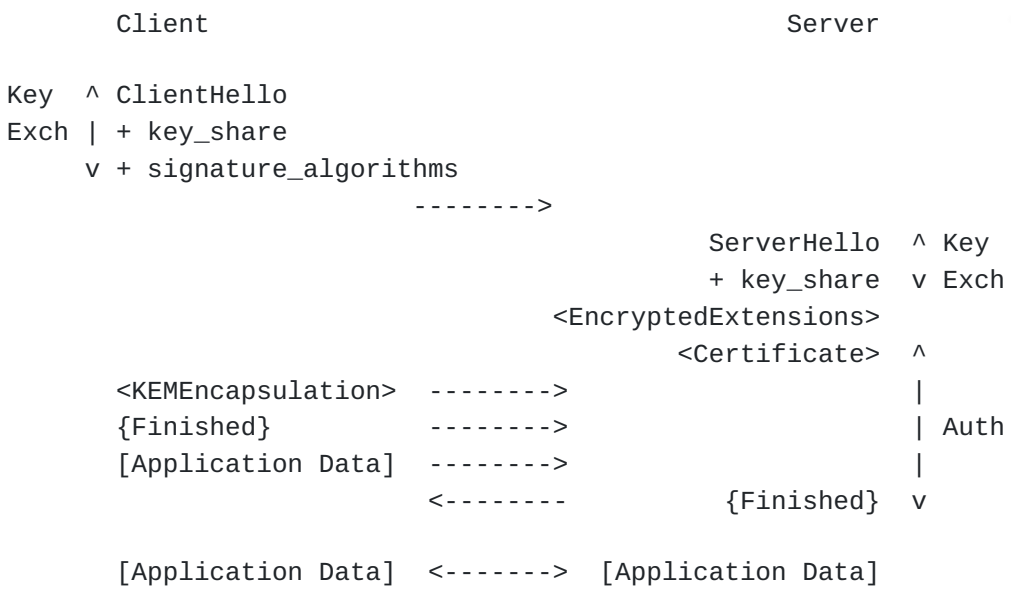
```
def Encapsulate(pk, context_string):
  enc, ctx = HPKE.SetupBaseS(pk, "tls13 auth-kem " + context_string)
  ss = ctx.Export("", HKDF.Length)
  return (enc, ss)

def Decapsulate(enc, sk, context_string):
  return HPKE.SetupBaseR(enc,
                         sk,
                         "tls13 auth-kem " + context_string)
             .Export("", HKDF.Length)
```

Keys are generated and encoded for transmission following the conventions in [RFC9180].

4. Full 1.5-RTT AuthKEM Handshake Protocol

Figure 1 below shows the basic KEM-authentication (KEM-Auth) handshake, without client authentication:



+ Indicates noteworthy extensions sent in the previously noted message.
<> 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 (KEM-Auth) Handshake without client authentication.

This basic handshake captures the core of AuthKEM. Instead of using a signature to authenticate the handshake, the client encapsulates a shared secret to the server's certificate public key. Only the server that holds the private key corresponding to the certificate public key can derive the same shared secret. This shared secret is mixed into the handshake's key schedule. The client does not have to wait for the server's Finished message before it can send data. The client knows that its message can only be decrypted if the server was able to derive the authentication shared secret encapsulated in the KEMEncapsulation message.

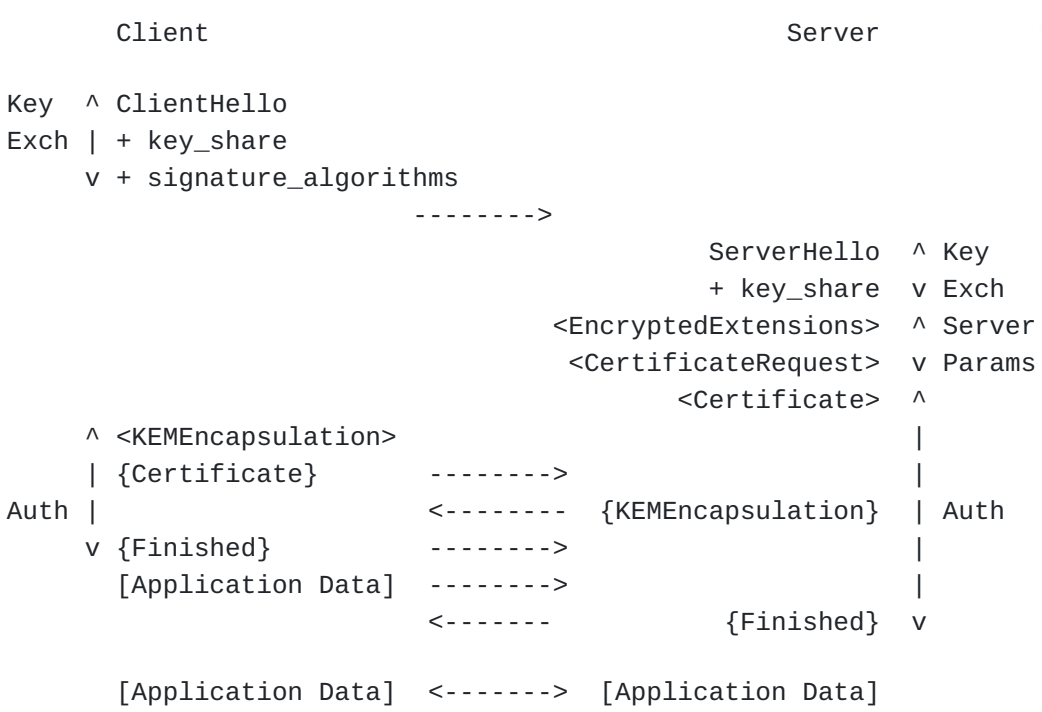
Finished messages are sent as in TLS 1.3, and achieve full explicit authentication.

4.1. Client authentication

For client authentication, the server sends the CertificateRequest message as in [RFC8446]. This message can not be authenticated in the AuthKEM handshake: we will discuss the implications below.

As in [RFC8446], section 4.4.2, if and only if the client receives CertificateRequest, it MUST send a Certificate message. If the client has no suitable certificate, it MUST send a Certificate message containing no certificates. If the server is satisfied with the provided certificate, it MUST send back a KEMEncapsulation message, containing the encapsulation to the client's certificate. The resulting shared secret is mixed into the key schedule. This ensures any messages sent using keys derived from it are covered by the authentication.

The AuthKEM handshake with client authentication is given in Figure 2.



+ Indicates noteworthy extensions sent in the previously noted message.

<> 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 2: Message Flow for KEM-Authentication (KEM-Auth) Handshake with client authentication.

If the server is not satisfied with the client's certificates, it MAY, at its discretion, decide to continue or terminate the handshake.

Unfortunately, AuthKEM client authentication requires an extra round-trip. Clients that know the server's long-term public KEM key MAY choose to use the abbreviated AuthKEM handshake and opportunistically send the client certificate as a 0-RTT-like message. We will discuss this later.

4.2. Relevant handshake messages

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 a CertificateRequest message (thus indicating that the client should not authenticate with a certificate). For AuthKEM, Certificate MUST include the long-term KEM public key. Certificates MUST be handled in accordance with [RFC8446], section 4.4.2.4.

  Certificates MUST be handled in accordance with [RFC8446], section 4.4.2.4.

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

4.3. Overview of key differences with RFC8446 TLS 1.3

* New types of signature_algorithms for KEMs.

* Public keys in certificates are KEM algorithms

* New handshake message KEMEncapsulation

* The key schedule mixes in the shared secrets from the authentication.

* The Certificate is sent encrypted with a new handshake encryption key.

* The client sends Finished before the server.

* The clients sends data before the server has sent Finished.

4.4. Implicit and explicit authentication

The data that the client MAY transmit to the server before having received the server's Finished is encrypted using ciphersuites chosen based on the client's and server's advertised preferences in the ClientHello and ServerHello messages. The ServerHello message can however not be authenticated before the Finished message from the server is verified. The full implications of this are discussed in the Security Considerations section.

Upon receiving the client's authentication messages, the server responds with its Finished message, which achieves explicit authentication. Upon receiving the server's Finished message, the client achieves explicit authentication. Receiving this message retroactively confirms the server's cryptographic parameter choices.

4.5. Authenticating CertificateRequest

The CertificateRequest message can not be authenticated during the AuthKEM handshake; only after the Finished message from the server has been processed, it can be proven as authentic. The security implications of this are discussed later.

This is dicussed in Github issue #16. We would welcome feedback there.

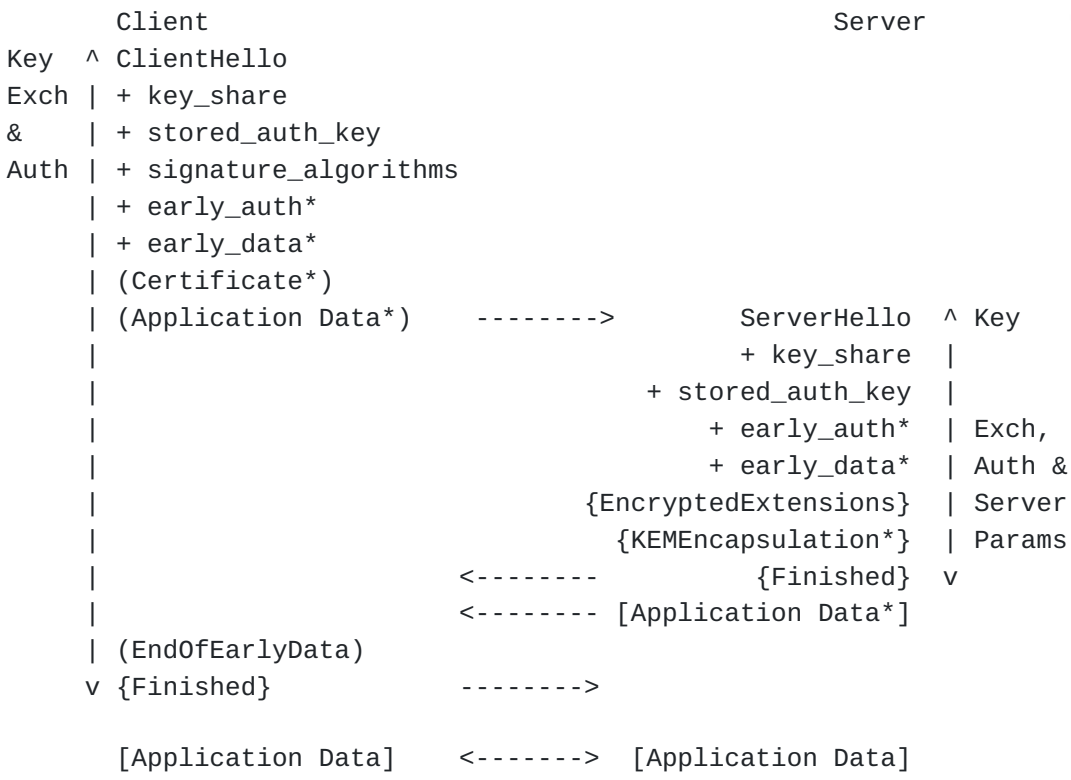
Clients MAY choose to only accept post-handshake authentication.

TODO: Should they indicate this? TLS Flag?

5. Abbreviated AuthKEM with pre-shared public KEM keys

When the client already has the server's long-term public key, we can do a more efficient handshake. The client will send the encapsulation to the server's long-term public key in a ClientHello extension. An overview of the abbreviated AuthKEM handshake is given in Figure 3.

A client that already knows the server, might also already know that it will be required to present a client certificate. This is expected to be especially useful in server-to-server scenarios. The abbreviated handshake allows to encrypt the certificate and send it like early data.



+ 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 client_early_handshake_traffic_secret.

() Indicates messages protected using keys derived from a client_early_traffic_secret.

{} Indicates messages protected using keys derived from a [sender]_handshake_traffic_secret.

[] Indicates messages protected using keys derived from [sender]_application_traffic_secret_N.

Figure 3: Abbreviated AuthKEM handshake, with optional opportunistic client authentication.

5.1. Negotiation

A client that knows a server's long-term KEM public key MAY choose to attempt the abbreviated AuthKEM handshake. If it does so, it MUST include the stored_auth_key extension in the ClientHello message. This message MUST contain the encapsulation against the long-term KEM public key. Details of the extension are described below. The shared secret resulting from the encapsulation is mixed in to the EarlySecret computation.

The client MAY additionally choose to send a certificate to the server. It MUST know what ciphersuites the server accepts before it does so. If it chooses to do so, it MUST send the early_auth extension to the server. The Certificate is encrypted with the client_early_handshake_traffic_secret.

The server MAY accept the abbreviated AuthKEM handshake. If it does, it MUST reply with a stored_auth_key extension. If it does not accept the abbreviated AuthKEM handshake, for instance because it does not have access to the correct secret key anymore, it MUST NOT reply with a stored_auth_key extension. The server, if it accepts the abbreviated AuthKEM handshake, MAY additionally accept the Certificate message. If it does, it MUST reply with a early_auth extension.

If the client, who sent a stored_auth_key extension, receives a ServerHello without stored_auth_key extension, it MUST recompute EarlySecret without the encapsulated shared secret.

If the client sent a Certificate message, it MUST drop that message from its transcript. The client MUST then continue with a full AuthKEM handshake.

5.2. 0-RTT, forward secrecy and replay protection

The client MAY send 0-RTT data, as in [RFC8446] 0-RTT mode. The Certificate MUST be sent before the 0-RTT data.

As the EarlySecret is derived only from a key encapsulated to a long-term secret, it does not have forward secrecy. Clients MUST take this into consideration before transmitting 0-RTT data or opting in to early client auth. Certificates and 0-RTT data may also be replayed.

This will be discussed in full under Security Considerations.

6. Implementation

In this section we will discuss the implementation details such as extensions and key schedule.

6.1. Negotiation of AuthKEM

Clients will indicate support for this mode by negotiating it as if it were a signature scheme (part of the signature_algorithms extension). We thus add these new signature scheme values (even though, they are not signature schemes) for the KEMs defined in [RFC9180] 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 AuthKEM support with the specified key exchange mode. These values MUST NOT appear in signature_algorithms_cert, as this extension specifies the signing algorithms by which certificates are signed.

6.2. ClientHello and ServerHello extensions

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

```
enum {
  ...
  stored_auth_key (TBD),                /* RFC TBD */
  early_auth (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
early_auth [RFCTBD]	CH, SH

6.2.1. Stored Auth Key

To transmit the early authentication encapsulation in the abbreviated AuthKEM handshake, this document defines a new extension type (stored_auth_key (TBD)). It is used in ClientHello and ServerHello messages.

The extension_data field of this extension, when included in the ClientHello, MUST contain the StoredInformation structure.

```
struct {
      select (type) {
        case client:
          opaque key_fingerprint<1..255>;
          opaque ciphertext<1..2^16-1>
        case server:
          AcceptedAuthKey '1';
      } body;
} StoredInformation
```

This extension MUST contain the folowing information when included in ClientHello messages:

- *The client indicates the public key encapsulated to by its fingerprint

- *The client submits the ciphertext

The server MUST send the extension back as an acknowledgement, if and only if it wishes to negotiated the abbreviated AuthKEM handshake.

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.

If this extension is not present, the client and the server MUST NOT negotiate the abbreviated AuthKEM handshake.

The presence of the fingerprint might reveal information about the identity of the server that the client has. This is discussed further under Security Considerations (Section 7).

6.2.2. Early authentication

To indicate the client will attempt client authentication in the abbreviated AuthKEM handshake, and for the server to indicate acceptance of attempting this authentication mechanism, we define the ```early_auth (TDB)`` extension. It is used in ClientHello and ServerHello messages.

```
struct {
} EarlyAuth
```

This is an empty extension.

It MUST NOT be sent if the stored_auth_key extension is not present.

6.3. Protocol messages

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 {
    ...
    kem_encapsulation(tbd),
    ...
    (255)
  } HandshakeType;

struct {
    HandshakeType msg_type;    /* handshake type */
    uint24 length;             /* remaining bytes in message */
    select (Handshake.msg_type) {
        ...
        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.

The KEMEncapsulation message is defined as follows:

```
struct {
    opaque certificate_request_context<0..2^8-1>
    opaque encapsulation<0..2^16-1>;
} KEMEncapsulation;
```

The encapsulation field is the result of a Encapsulate function. The Encapsulate() function will also result in a shared secret (ssS or ssC, depending on the peer) 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 Decapsulate(enc, skR) operation by using the sent encapsulation and the private key of the public key advertised in the end-entity certificate sent. The Decapsulate(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.

certificate_request_context is included to allow the recipient to identify the certificate against which the encapsulation was generated. It MUST be set to the value in the Certificate message to which the encapsulation was computed.

6.4. Cryptographic computations

The AuthKEM 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.

6.4.1. Key schedule for full AuthKEM handshakes

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

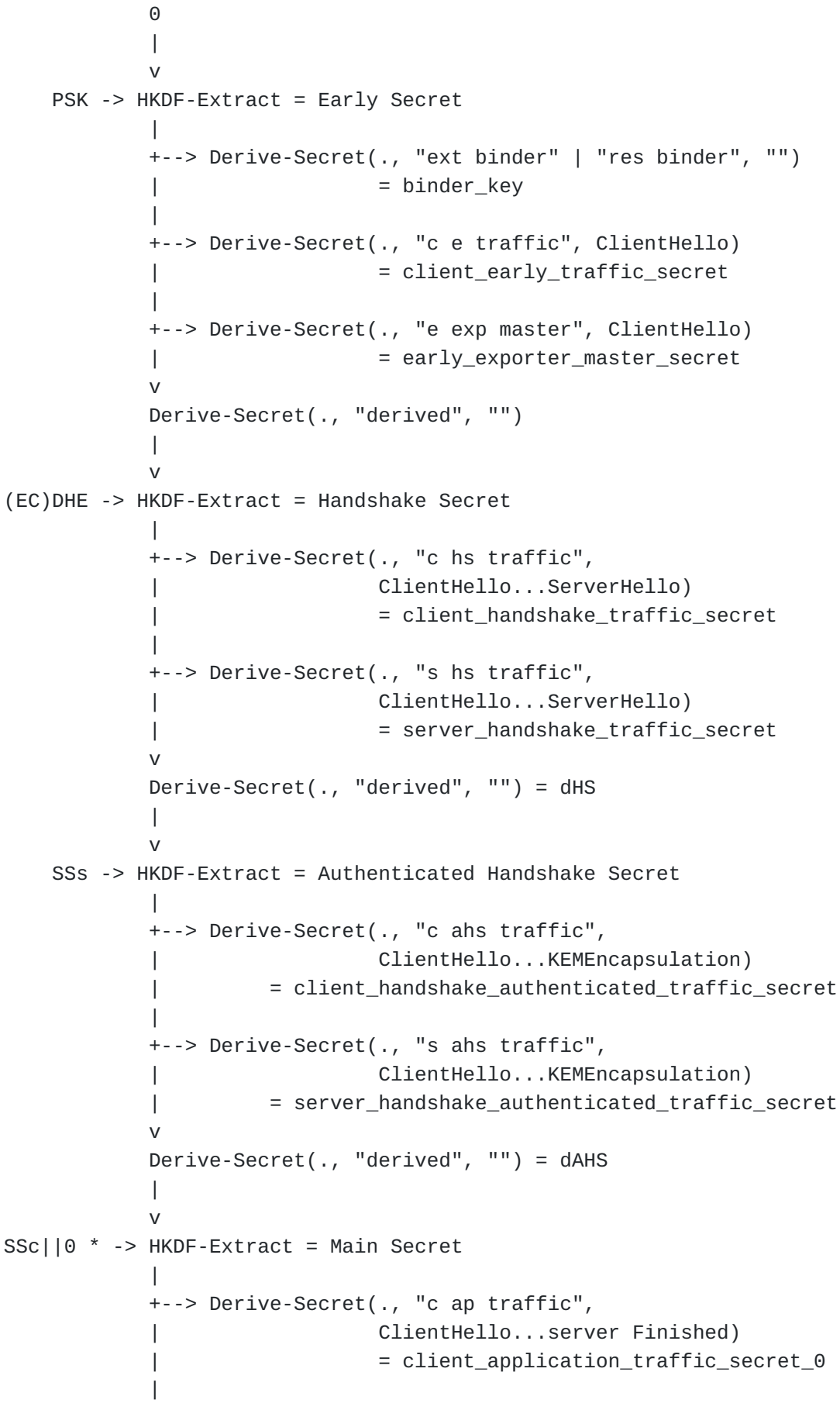
Keys are derived from two input secrets using the HKDF-Extract and Derive-Secret functions. The general pattern for adding a new secret is to use HKDF-Extract with the Salt being the current secret state and the Input Keying Material (IKM) being the new secret to be added.

The notable differences are:

- The addition of the Authenticated Handshake Secret and a new set of handshake traffic encryption keys.

- The inclusion of the SSs and SSc shared secrets as IKM to Authenticated Handshake Secret and Main Secret, respectively

The full key schedule proceeds as follows:


0
|
v
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
v
Derive-Secret(., "derived", "")
|
v
(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
v
Derive-Secret(., "derived", "") = dHS
|
v
SSs -> HKDF-Extract = Authenticated Handshake Secret
|
+--> Derive-Secret(., "c ahs traffic",
| ClientHello...KEMEncapsulation)
| = client_handshake_authenticated_traffic_secret
|
+--> Derive-Secret(., "s ahs traffic",
| ClientHello...KEMEncapsulation)
| = server_handshake_authenticated_traffic_secret
v
Derive-Secret(., "derived", "") = dAHS
|
v
SSc||0 * -> HKDF-Extract = Main 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
```

*: if client authentication was requested, the `SSc` value should be used. Otherwise, the `0` value is used.

6.4.2. Abbreviated AuthKEM key schedule

The abbreviated AuthKEM handshake follows the [RFC8446] key schedule more closely. We change the computation of the EarlySecret as follows, and add a computation for client_early_handshake_traffic_secret: ~~~ 0 | v SSs -> HKDF-Extract = Early Secret | ... +--> Derive-Secret(., "c e traffic", ClientHello) | = client_early_traffic_secret | +--> Derive-Secret(., "c e hs traffic", ClientHello) | = client_early_handshake_traffic_secret ... ~~~

We change the computation of Main Secret as follows: ~~~ Derive-Secret(., "derived", "") = dHS | v SSc||0 * -> HKDF-Extract = Main Secret | ... ~~~

6.4.3. Computations of KEM shared secrets

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

```
SSs, encapsulation <- Encapsulate(public_key_server,
                                  "server authentication")
               SSc <- Decapsulate(encapsulation, private_key_client,
                                  "client authentication")
```

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

```
               SSs <- Decapsulate(encapsulation, private_key_server
                                  "server authentication")
SSc, encapsulation <- Encapsulate(public_key_client,
                                  "client authentication")
```

6.4.4. Explicit Authentication Messages

As discussed, AuthKEM generally uses a message for explicit authentication: Finished message. Note that in the full handshake, AuthKEM 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 abbreviated handshake 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 AuthKEM handshake completes.

The key used to compute the Finished message MUST be computed from the MainSecret using HKDF. Specifically:

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

See the abbreviated AuthKEM handshake negotiation section (Section 5.1) for special considerations for the abbreviated AuthKEM handshake.

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. Security Considerations

*The academic works proposing AuthKEM (KEMTLS) 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 (the abbreviated AuthKEM handshake) has a proof for both unilaterally and mutually authenticated handshakes.

*We have proofs of the security of KEMTLS and KEMTLS-PDK in Tamarin. The academic write-up of this is work in progress.

*Application Data sent prior to receiving the server's last explicit authentication message (the Finished message) can be subject to a client certificate suite downgrade attack. 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.

* The PDK extension identifies the public key to which the client has encapsulated via a hash. This reveals some information about which server identity the client has. [I-D.ietf-tls-esni-14] may help alleviate this.

7.1. 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.

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, being sent to an implicitly authenticated peer.

7.2. Authentication of Certificate Request

Due to the implicit authentication of the server's messages during the full AuthKEM handshake, the CertificateRequest message can not be authenticated before the client received Finished.

The key schedule guarantees that the server can not read the client's certificate message (as discussed above). An active adversary that maliciously inserts a CertificateRequest message will also result in a mismatch in transcript hashes, which will cause the handshake to fail.

However, there may be side effects. The adversary might learn that the client has a certificate by observing the length of the messages sent. There may also be side-effects, especially in situations where the client is prompted to e.g. approve use or unlock a certificate stored encrypted or on a smart card.

8. References

8.1. Normative References

[I-D.ietf-tls-esni-14] Rescorla, E., Oku, K., Sullivan, N., and C. A. Wood, "TLS Encrypted Client Hello", Work in Progress, Internet-Draft, draft-ietf-tls-esni-14, 13 February 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-esni-14>.

[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-11, 23 September 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-subcerts-11>.

[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>.

[RFC9180] Barnes, R., Bhargavan, K., Lipp, B., and C. Wood, "Hybrid Public Key Encryption", RFC 9180, DOI 10.17487/RFC9180, February 2022, <https://www.rfc-editor.org/rfc/rfc9180>.

8.2. 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] Stebil, D., Schwabe, P., and T. Wiggers, "More Efficient KEMTLS with Pre-Shared Keys", DOI 10.1007/978-3-030-88418-5_1, IACR ePrint https://ia.cr/2021/779, May 2021, <https://doi.org/10.1007/978-3-030-88418-5_1>.

[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.

[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).

Appendix B. Open points of discussion

The following are open points for discussion. The corresponding Github issues will be linked.

B.1. Authentication concerns for client authentication requests.

Tracked by Issue #16.

The certificate request message from the server can not be authenticated by the AuthKEM mechanism. This is already somewhat discussed above and under security considerations. We might want to allow clients to refuse client auth for scenarios where this is a concern.

B.2. Interaction with signing certificates

Tracked by Issue #20.

In the current state of the draft, we have not yet discussed combining traditional signature-based authentication with KEM-based authentication. One might imagine that the Client has a sigining certificate and the server has a KEM public key.

In the current draft, clients MUST use a KEM certificate algorithm if the server negotiated AuthKEM.

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