TLS Working Group
Internet-Draft
Intended status: Informational
Expires: 17 October 2024

T. Wiggers
PQShield
S. Celi
Brave Software
P. Schwabe
Radboud University and MPI-SP
D. Stebila
University of Waterloo
N. Sullivan
15 April 2024

KEM-based Authentication for TLS 1.3
draft-celi-wiggers-tls-authkem-03

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.

About This Document

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

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-celi-wiggers-tls-authkem/.

Discussion of this document takes place on the tlswg Working Group mailing list (mailto:tls@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/tls/. Subscribe at https://www.ietf.org/mailman/listinfo/tls/.

Source for this draft and an issue tracker can be found at https://github.com/kemtls/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/.

Copyright Notice

Table of Contents

1. Introduction

Note: This is a work-in-progress draft. We welcome discussion, feedback and contributions through the IETF TLS working group mailing list or directly on GitHub. Any code points indicated by this draft are for experiments only, and should not be expected to be stable.

This document gives a construction for KEM-based authentication in TLS 1.3 [RFC8446]. 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 its peer's 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 build on [RFC9180]. This standard currently only covers Diffie-Hellman based KEMs, but the first post-quantum algorithms have already been put forward [I-D.draft-westerbaan-cfrg-hpke-xyber768d00]. This proposal uses Kyber [KYBER] [I-D.draft-cfrg-schwabe-kyber], the first selected algorithm for key exchange in the NIST post-quantum standardization project [NISTPQC].

1.1. Revision history

This section should be removed prior to publication of a final version of this document.

* Revision draft-celi-wiggers-tls-authkem-03
  - Assigned experimental code points
  - Re-worked HPKE computation
* Revision draft-celi-wiggers-tls-authkem-02
  - Split PSK mechanism off into [I-D.draft-wiggers-tls-authkem-psk]
  - Editing
* Revision draft-celi-wiggers-tls-authkem-01
  - Significant Editing
  - Use HPKE context
* Revision draft-celi-wiggers-tls-authkem-00
  - Initial version

1.1.1. Revision 2

1.2. Using key exchange instead of signatures for authentication

The elliptic-curve and finite-field-based key exchange and signature algorithms that are currently widely used are very similar in sizes for public keys, ciphertexts and signatures. As an example, RSA signatures are famously "just" RSA encryption backwards.

This changes in the post-quantum setting. Post-quantum key exchange and signature algorithms have significant differences in implementation, performance characteristics, and key and signature sizes.

This also leads to increases in code size: For example, implementing highly efficient polynomial multiplication for post-quantum KEM Kyber and signature scheme Dilithium [DILITHIUM] requires significantly different approaches, even though the algorithms are related [K22].

Using the protocol proposed in this draft allows to reduce the amount of data exchanged for handshake authentication. It also allows to re-use the implementation that is used for ephemeral key exchange for authentication, as KEM operations replace signing. This decreases the code size requirements, which is especially relevant to protected implementations. Finally, KEM operations may be more efficient than signing, which might especially affect embedded platforms.

1.3. Evaluation of handshake sizes

Should probably be removed before publishing

In the following table, we compare the sizes of TLS 1.3- and AuthKEM-based handshakes. We give the transmission requirements for handshake authentication (public key + signature), and certificate chain (intermediate CA certificate public key and signature + root CA signature). For clarity, we are not listing post-quantum/traditional hybrid algorithms; we also omit mechanisms such as Certificate Transparency [RFC6962] or OCSP stapling [RFC6960]. We use Kyber-768 instead of the smaller Kyber-512 parameter set, as the former is currently used in experimental deployments. For signatures, we use Dilithium, the "primary" algorithm selected by NIST for post-quantum signatures, as well as Falcon [FALCON], the algorithm that offers smaller public key and signature sizes, but which NIST indicates can be used if the implementation requirements can be met.

Handshake	HS auth algorithm	HS Auth bytes	Certificate chain bytes	Sum
TLS 1.3	RSA-2048	528	784 (RSA-2048)	1312
TLS 1.3	Dilithium-2	3732	6152 (Dilithium-2)	9884
TLS 1.3	Falcon-512	1563	2229 (Falcon-512)	3792
TLS 1.3	Dilithium-2	3732	2229 (Falcon-512)	5961
AuthKEM	Kyber-768	2272	6152 (Dilithium-2)	8424
AuthKEM	Kyber-768	2272	2229 (Falcon-512)	4564

Table 1: Size comparison of public-key cryptography in TLS 1.3 and AuthKEM handshakes.

Note that although TLS 1.3 with Falcon-512 is the smallest instantiation, Falcon is very challenging to implement: signature generation requires (emulation of) 64-bit floating point operations in constant time. It is also very difficult to protect against other side-channel attacks, as there are no known methods of masking Falcon. In light of these difficulties, use of Falcon-512 in online handshake signatures may not be wise.

Using AuthKEM with Falcon-512 in the certificate chain remains an attractive option, however: the certificate issuance process, because it is mostly offline, could perhaps be set up in a way to protect the Falcon implementation against attacks. Falcon signature verification is fast and does not require floating-point arithmetic. Avoiding online usage of Falcon in TLS 1.3 requires two implementations of the signature verification routines, i.e., Dilithium and Falcon, on top of the key exchange algorithm.

In all examples, the size of the certificate chain still dominates the TLS handshake, especially if Certificate Transparency SCT statements are included, which is relevant in the context of the WebPKI. However, we believe that if proposals to reduce transmission sizes of the certificate chain in the WebPKI context are implemented, the space savings of AuthKEM naturally become relatively larger and more significant. We discuss this in Section 1.4.2.

1.4. Related work

1.4.1. OPTLS

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.

1.4.2. Compressing certificates and certificate chains

AuthKEM reduces the amount of data required for authentication in TLS. In recognition of the large increase in handshake size that a naive adoption of post-quantum signatures would affect, several proposals have been put forward that aim to reduce the size of certificates in the TLS handshake. [RFC8879] proposes a certificate compression mechanism based on compression algorithms, but this is not very helpful to reduce the size of high-entropy public keys and signatures. Proposals that offer more significant reductions of sizes of certificate chains, such as [I-D.draft-jackson-tls-cert-abridge], [I-D.ietf-tls-ctls], [I-D.draft-kampanakis-tls-scas-latest], and [I-D.draft-davidben-tls-merkle-tree-certs] all mainly rely on some form of out-of-band distribution of intermediate certificates or other trust anchors in a way that requires a robust update mechanism. This makes these proposals mainly suitable for the WebPKI setting; although this is also the setting that has the largest number of certificates due to the inclusion of SCT statements [RFC6962] and OSCP staples [RFC6960].

AuthKEM complements these approaches in the WebPKI setting. On its own the gains that AuthKEM offers may be modest compared to the large sizes of certificate chains. But when combined with compression or certificate suppression mechanisms such as those proposed in the referenced drafts, the reduction in handshake size when replacing Dilithium-2 by Kyber-768 becomes significant again.

1.5. Organization

After a brief introduction to KEMs, we will introduce the AuthKEM authentication mechanism. For clarity, we discuss unilateral and mutual authentication separately. 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. The draft concludes with ah extensive discussion of relevant security considerations.

A related mechanism for KEM-based PSK-style handshakes is discussed in [I-D.draft-wiggers-tls-authkem-psk].

2. Conventions and definitions

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.

2.1. 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.

2.2. 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, per HPKE [RFC9180]:

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

Decapsulate(enc, skR, context_string):
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")
  ss = ctx.Export(context_string, HKDF.Length)
  return (enc, ss)

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

Keys are generated and encoded for transmission following the conventions in [RFC9180]. The values of context_string are defined in Section 4.3.2.

Open question: Should we keep using HPKE, or just use "plain" KEMs, as in the original KEMTLS works? Please see the discussion at Issue #32 (https://github.com/kemtls/draft-celi-wiggers-tls-authkem/issues/32).

3. Full 1.5-RTT AuthKEM Handshake Protocol

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

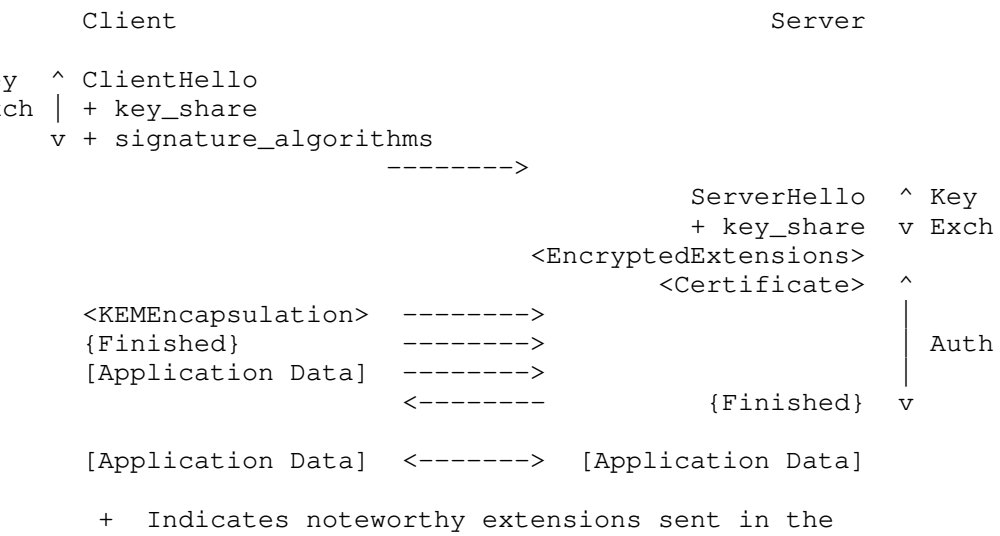


Figure 1: Message Flow for KEM-Authentication (KEM-Auth) Handshake without client authentication.

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.

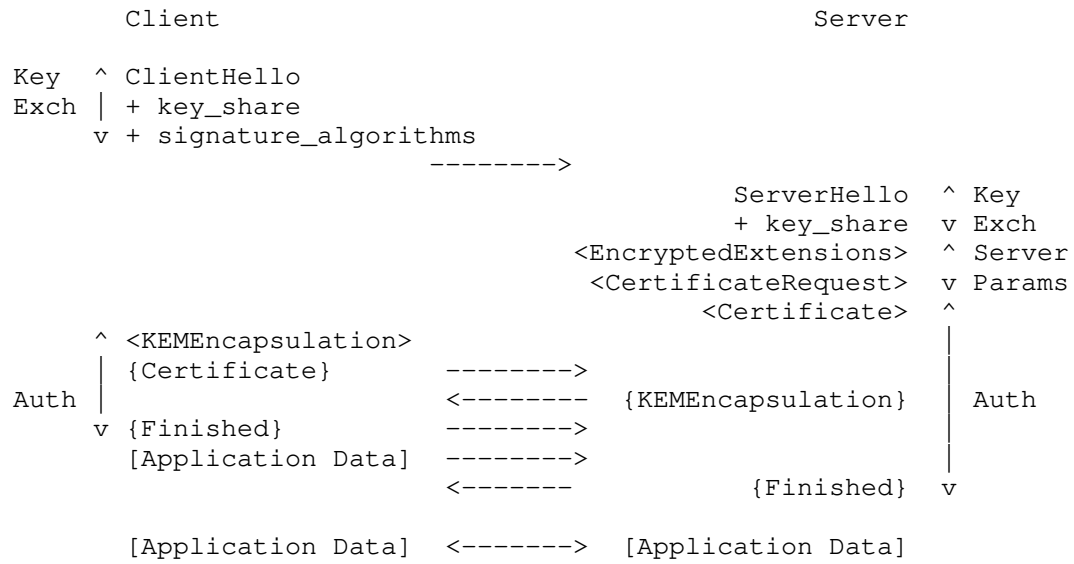
Finished messages are sent as in TLS 1.3, and achieve full explicit authentication.

3.1. 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. If it decides to continue, it MUST NOT send back a KEMEncapsulation message and the client and server MUST compute the encryption keys as in the server-only authenticated AuthKEM handshake. The Certificate remains included in the transcript. The client MUST NOT assume it has been authenticated.

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. This mechanism is discussed in [I-D.draft-wiggers-tls-authkem-psk].

3.2. 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 MUST be 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.

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

3.3. 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 client sends data before the server has sent Finished.

3.4. 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.

3.5. 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 discussed in GitHub issue #16 (https://github.com/kemtls/draft-celi-wiggers-tls-authkem/issues/16). We would welcome feedback there.

Clients MAY choose to only accept post-handshake authentication.

TODO: Should they indicate this? TLS Flag?

4. Implementation

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

4.1. 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 => 0xFE01,
  dhkem_x448_sha512   => TBD,
  kem_x25519kyber768  => TBD, /*draft-westerbaan-cfrg-hpke-xyber768d00*/
}
```

Please give feedback on which KEMs should be included

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.

4.2. 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(30),
    ...
    (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 a "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 (See Section 4.3.1).

If the KEMEncapsulation message is sent by a server, the authentication algorithm MUST be one offered in the client's signature_algorithms extension. Otherwise, the server MUST terminate the handshake with an "unsupported_certificate" alert.

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() operation by using the sent encapsulation and the private key of the public key advertised in the end-entity certificate sent. The Decapsulate() 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.

4.3. 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.

4.3.1. 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].

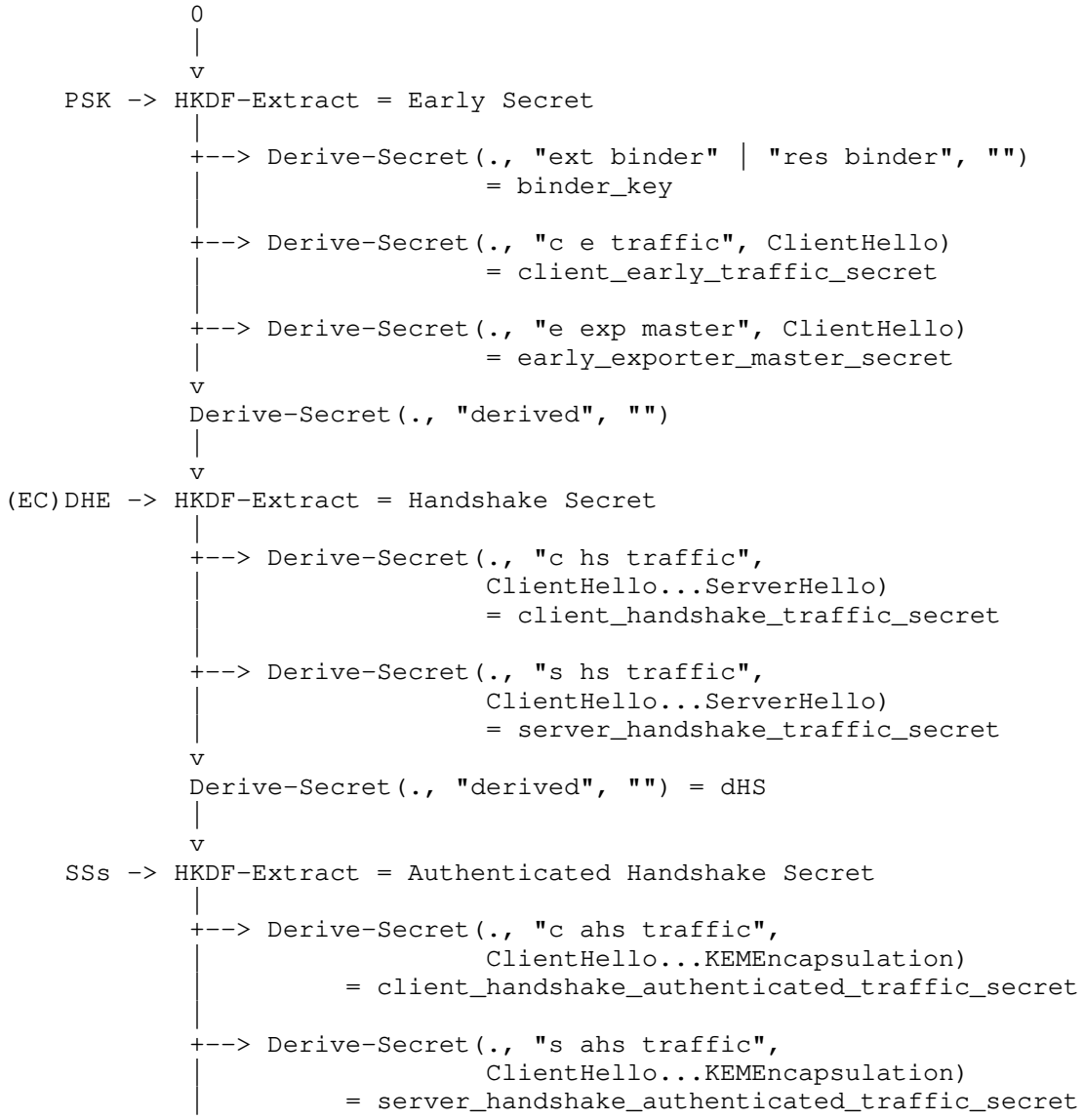
Keys are derived from two input secrets using the HKDF-Extract and Derive-Secret functions. The general pattern for adding a new secret is to use HKDF-Extract with the Salt being the current secret state and the Input Keying Material (IKM) being the new secret to be added.

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 (if present) shared secrets as IKM to Authenticated Handshake Secret and Main Secret, respectively.

The full key schedule proceeds as follows:



```
                v
                Derive-Secret(., "derived", "") = dAHS
                |
                v
   SSc||0 * -> HKDF-Extract = Main Secret
                |
                +--> Derive-Secret(., "c ap traffic",
                |                  ClientHello...client 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...server Finished)
                                   = resumption_master_secret
```

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

4.3.2. 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")
```

4.3.3. Explicit Authentication Messages

AuthKEM upgrades implicit to explicit authentication through the 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).

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 (instead of a key derived from HS as in [RFC8446]). 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. Note that instead of what is specified in [RFC8446], we use the full transcript for both server and client Finished messages:

```
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 [RFC8446]. In particular, this includes any alerts sent by the server in response to client Certificate and KEMEncapsulation messages.

See [SSW20] for a full treatment of implicit and explicit authentication.

5. Security Considerations

5.1. 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.

5.2. 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.

5.3. Other security considerations

* Because the Main Secret is derived from both the ephemeral key exchange, as well as from the key exchanges completed for server and (optionally) client authentication, the MS secret always reflects the peers' views of the authentication status correctly. This is an improvement over TLS 1.3 for client authentication.

* The academic works proposing AuthKEM (KEMTLS) contains an in-depth technical discussion of and a proof of the security of the handshake protocol without client authentication [SSW20], [Wig24].

* The work proposing the variant protocol [SSW21], [Wig24] 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. [CHSW22]

* 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.

* If AuthKEM client authentication is used, the resulting shared secret is included in the key schedule. This ensures that both peers have a consistent view of the authentication status, unlike [RFC8446].

6. References

6.1. 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/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>.

[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>.

6.2. Informative References

[CHSW22] Celi, S., Hoyland, J., Stebila, D., and T. Wiggers, "A tale of two models: formal verification of KEMTLS in Tamarin", ESORICS 2022 , DOI 10.1007/978-3-031-17143-7_4, IACR ePrint https://ia.cr/2022/1111, August 2022, <https://doi.org/10.1007/978-3-031-17143-7_4>.

[DILITHIUM] Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., and D. Stehlé, "CRYSTALS-Dilithium", 2021, <https://pq-crystals.org/dilithium/>.

[FALCON] Fouque, P., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Ricosset, T., Seiler, G., Whyte, W., and Z. Zhang, "Falcon", 2021, <https://falcon-sign.info>.

[I-D.draft-cfrg-schwabe-kyber] Schwabe, P. and B. Westerbaan, "Kyber Post-Quantum KEM", Work in Progress, Internet-Draft, draft-cfrg-schwabe-kyber-04, 2 January 2024, <https://datatracker.ietf.org/doc/html/draft-cfrg-schwabe-kyber-04>.

[I-D.draft-davidben-tls-merkle-tree-certs] Benjamin, D., O'Brien, D., and B. Westerbaan, "Merkle Tree Certificates for TLS", Work in Progress, Internet-Draft, draft-davidben-tls-merkle-tree-certs-02, 4 March 2024, <https://datatracker.ietf.org/doc/html/draft-davidben-tls-merkle-tree-certs-02>.

[I-D.draft-jackson-tls-cert-abridge] Jackson, D., "Abridged Compression for WebPKI Certificates", Work in Progress, Internet-Draft, draft-jackson-tls-cert-abridge-00, 6 July 2023, <https://datatracker.ietf.org/doc/html/draft-jackson-tls-cert-abridge-00>.

[I-D.draft-kampanakis-tls-scas-latest]
Kampanakis, P., Bytheway, C., Westerbaan, B., and M. Thomson, "Suppressing CA Certificates in TLS 1.3", Work in Progress, Internet-Draft, draft-kampanakis-tls-scas-latest-03, 5 January 2023, <https://datatracker.ietf.org/doc/html/draft-kampanakis-tls-scas-latest-03>.

[I-D.draft-westerbaan-cfrg-hpke-xyber768d00]
Westerbaan, B. and C. A. Wood, "X25519Kyber768Draft00 hybrid post-quantum KEM for HPKE", Work in Progress, Internet-Draft, draft-westerbaan-cfrg-hpke-xyber768d00-02, 4 May 2023, <https://datatracker.ietf.org/doc/html/draft-westerbaan-cfrg-hpke-xyber768d00-02>.

[I-D.draft-wiggers-tls-authkem-psk]
Wiggers, T., Celi, S., Schwabe, P., Stebila, D., and N. Sullivan, "KEM-based pre-shared-key handshakes for TLS 1.3", Work in Progress, Internet-Draft, draft-wiggers-tls-authkem-psk-00, 18 August 2023, <https://datatracker.ietf.org/doc/html/draft-wiggers-tls-authkem-psk-00>.

[I-D.ietf-tls-ctls]
Rescorla, E., Barnes, R., Tschofenig, H., and B. M. Schwartz, "Compact TLS 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-ctls-09, 23 October 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-ctls-09>.

[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 and DTLS", Work in Progress, Internet-Draft, draft-ietf-tls-subcerts-15, 30 June 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-subcerts-15>.

[K22]
Kannwischer, M. J., "Polynomial Multiplication for Post-Quantum Cryptography", Ph.D. thesis, 22 April 2022, <https://kannwischer.eu/thesis/>.

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

[KYBER] Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J., Schwabe, P., Seiler, G., and D. Stehlé, "CRYSTALS-Kyber", 2021, <https://pq-crystals.org/kyber/>.

[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>.

[RFC6960] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 6960, DOI 10.17487/RFC6960, June 2013, <https://www.rfc-editor.org/rfc/rfc6960>.

[RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <https://www.rfc-editor.org/rfc/rfc6962>.

[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>.

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

[SSW20] Stebila, D., Schwabe, P., and T. Wiggers, "Post-Quantum TLS without Handshake Signatures", ACM CCS 2020 , DOI 10.1145/3372297.3423350, IACR ePrint https://ia.cr/2020/534, November 2020, <https://doi.org/10.1145/3372297.3423350>.

[SSW21] Stebila, D., Schwabe, P., and T. Wiggers, "More Efficient KEMTLS with Pre-Shared Keys", ESORICS 2021 , 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>.

[Wig24] Wiggers, T., "Post-Quantum TLS", PhD thesis https://thomwiggers.nl/publication/thesis/, 9 January 2024.

Appendix A. Open points of discussion

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

A.1. Authentication concerns for client authentication requests.

Tracked by Issue #16 (https://github.com/kemtls/draft-celi-wiggers-tls-authkem/issues/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.

A.2. Interaction with signing certificates

Tracked by Issue #20 (https://github.com/kemtls/draft-celi-wiggers-tls-authkem/issues/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.

Acknowledgements

Early versions of this work were supported by the European Research Council through Starting Grant No. 805031 (EPOQUE).

Part of this work was supported by the NLNet NGI Assure theme fund project "Standardizing KEMTLS" (https://nlnet.nl/project/KEMTLS/)

Authors' Addresses

Thom Wiggers
PQShield
Nijmegen
Email: thom@thomwiggers.nl

Sofía Celi
Brave Software
Lisbon
Portugal
Email: cherenkov@riseup.net

Peter Schwabe
Radboud University and MPI-SP
Email: peter@cryptojedi.org

Douglas Stebila
University of Waterloo
Waterloo, ON
Canada
Email: dstebila@uwaterloo.ca

Nick Sullivan
Email: nicholas.sullivan+ietf@gmail.com