Workgroup: TLS Working Group
Internet-Draft:
draft-wiggers-tls-authkem-psk-00
Published: 18 August 2023
Intended Status: Informational
Expires: 19 February 2024
Authors: T. Wiggers S. Celi
PQShield Brave Software
P. Schwabe
Radboud University and MPI-SP
D. Stebila N. Sullivan
University of Waterloo

KEM-based pre-shared-key handshakes 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.

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-wiggers-tls-authkem-psk/.

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/.

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 19 February 2024.

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.

This document gives a construction for KEM-based, PSK-style abbreviated TLS 1.3 [RFC8446] handshakes. It is similar in spirit to [I-D.draft-celi-wiggers-tls-authkem], but can be independently implemented.

The abbreviated handshake is appropriate for endpoints that have KEM public keys, and where the client has the server's public key before initiation of the connection. 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]. The public keys need not necessarily be certificates, however. The client might be provided with the public key as a matter of configuration.

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. Related work

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.

1.2. Organization

After covering preliminaries, we introduce the abbreviated AuthKEM-PSK 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. 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.

This definition matches the one from [I-D.draft-celi-wiggers-tls-authkem].

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_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 key schedule in TLS 1.3:

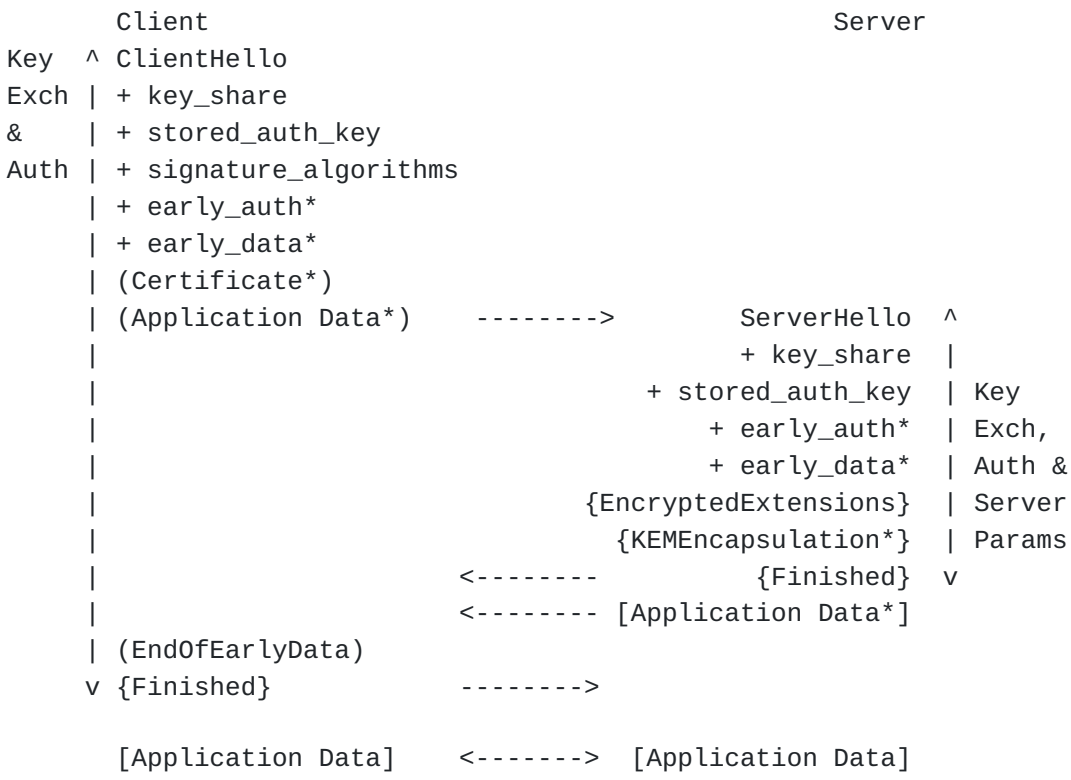
```
def Encapsulate(pk, context_string):
  enc, ctx = HPKE.SetupBaseS(pk,
                             "tls13 auth-kem " + context_string)
  ss = ctx.Export("", HKDF.Length)
  return (enc, ss)

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]. The values of context_string are defined in Section 4.4.2.

3. 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 similarly to 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.

3.1. Negotiation

In an Appendix A.1, we sketch a variant based on the PSK extension.

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.

3.2. 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.

4. Implementation

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

4.1. Negotiation of AuthKEM algorithms

Clients and servers indicate support for AuthKEM authentication 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,
  kem_x25519kyber768  => TBD, /*draft-westerbaan-cfrg-hpke-xyber768d00*/
}
```

This matches the definition in [I-D.draft-celi-wiggers-tls-authkem].

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. 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

4.2.1. 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 following 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 negotiate 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 5).

4.2.2. 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 inClientHelloandServerHello`` messages.

```
struct {} EarlyAuth
```

This is an empty extension.

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

4.3. 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, except for the addition of a KEMEncapsulation message and does not use the CertificateVerify one.

Note that these definitions mirror [I-D.draft-celi-wiggers-tls-authkem].

```
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 3. 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() 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.

4.4. 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.

4.4.1. AuthKEM-PSK key schedule

The AuthKEM-PSK handshake follows the [RFC8446] key schedule 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
            ...
            |
            v
            Derive-Secret(., "derived", "") = dES
            ...
```

We change the computation of Main Secret as follows:

```
            Derive-Secret(., "derived", "") = dHS
            |
            v
SSc||0 * -> HKDF-Extract = Main Secret
            |
            ...
```

SSc is included if client authentication is used; otherwise, the value 0 is used.

4.4.2. Computations of KEM shared secrets

As in [I-D.draft-celi-wiggers-tls-authkem], 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")
```

4.4.3. Explicit Authentication Messages

AuthKEM upgrades implicit to explicit authentication through the Finished message. With AuthKEM-PSK, the server achieves explicit

authentication when sending their Finished message and the client when they send their Finished message.

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.

These computations match [I-D.draft-celi-wiggers-tls-authkem].

See Section 3.1 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.

5. 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].

*The work proposing the variant protocol [SSW21] with pre-distributed public keys (the abbreviated AuthKEM handshake) has a proof for both unilaterally and mutually authenticated handshakes.

*We have machine-verified proofs of the security of KEMTLS and KEMTLS-PDK in Tamarin. [CHSW22]

*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.

*If AuthKEM-PSK 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].

5.1. Server Anonymity

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] may help alleviate this.

An alternative approach could be the use of trial decryption. If the KEM used has anonymity, the ciphertext that the client sends is not linkable to the server public key. Kyber offers post-quantum anonymity [MX22].

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>.

[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>.

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/>.

[I-D.draft-celi-wiggers-tls-authkem] Celi, S., Schwabe, P., Stebila, D., Sullivan, N., and T. Wiggers, "KEM-based Authentication for TLS 1.3", Work in Progress, Internet-Draft, draft-celi-wiggers-tls-authkem-01, 7 March 2022, <https://datatracker.ietf.org/doc/html/draft-celi-wiggers-tls-authkem-01>.

[I-D.draft-cfrg-schwabe-kyber] Schwabe, P. and B. Westerbaan, "Kyber Post-Quantum KEM", Work in Progress, Internet-Draft, draft-cfrg-schwabe-kyber-02, 31 March 2023, <https://datatracker.ietf.org/doc/html/draft-cfrg-schwabe-kyber-02>.

[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.ietf-tls-esni] Rescorla, E., Oku, K., Sullivan, N., and C. A. Wood, "TLS Encrypted Client Hello", Work in Progress, Internet-Draft, draft-ietf-tls-esni-16, 6 April 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-esni-16>.

[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>.

[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/>.

[MX22] Maram, V. and K. Xagawa, "Post-Quantum Anonymity of Kyber", PKC 2023, IACR ePrint https://ia.cr/2022/1696, 2022.

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

[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>.

Appendix A. Open points of discussion

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

A.1. Alternative implementation based on the pre_shared_key extension

This is discussed in Issue #25.

[RFC8446] defines a PSK handshake that can be used with symmetric keys from e.g. session tickets. In this section, we sketch an alternative approach to AuthKEM-PSK based on the pre_shared_key extension.

A client needs to be set up with the following information:

```
struct {
    uint32 authkem_psk_config_version;
    uint32 config_lifetime;
    opaque KEMPublicKey;
} AuthKEMPSKConfig;
```

The client computes a KEM ciphertext and shared secret as follows:

```
SSs, encapsulation <- Encapsulate(public_key_server,
                                  "server authentication")
```

SSs is used in place of PSK in the TLS 1.3 key schedule, and binder_key is derived as follows:

```
          0
          |
          v
SSc ->  HKDF-Extract = Early Secret
          |
          +-----> Derive-Secret(., "ext binder" | "res binder", "")
          |                     = binder_key
          ...
```

In the pre_shared_key extension's identities, the client sends the following data:

```
struct {
  uint32 authkem_psk_config_version;
  opaque KEMCiphertext;
} AuthKEMPSKIdentity
```

The server computes the shared secret SSs from AuthKEMPSKIdentity.KEMCiphertext as follows:

```
SSs <- Decapsulate(encapsulation,
                   private_key_server
                   "server authentication")
```

The PSK binder value is computed as specified in [RFC8446], section 4.2.11.2. The server MUST verify the binder before continuing and abort the handshake if verification fails.

To be determined: how to handle immediate client authentication.

A.2. Interactions with DTLS

It is currently open if there need te be made modifications to better support integration with DTLS. Discussion is at Issue #23.

A.3. 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 signing 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

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

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