Internet Engineering Task Force
Internet-Draft
Intended status: Standards Track
Expires: 29 August 2025

W. Guo
L. Xia
J. Li
Huawei Technologies

25 February 2025

Post-Handshake Authentication via PAKE for TLS 1.3
draft-guo-pake-pha-tls-01

Abstract

This document provides a mechanism that uses password-authenticated key exchange (PAKE) as a post-handshake authentication for TLS 1.3, and that supports PAKE algorithms negotiation and optional channel binding.

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 29 August 2025.

Table of Contents

1. Introduction 2
2. Terminology 3
3. Post-Handshake Authentication via PAKE for TLS 1.3 3
3.1. PAKE Handshake Messages 3
3.1.1. PAKE Client Hello 5
3.1.2. PAKE Server Hello 5
3.1.3. PAKE Hello Retry Request 6
3.1.4. PAKE Finished 6
3.1.5. PAKE Status 6
3.2. Channel Binding 7
3.3. PAKE-based Post-Handshake Authentication 8
4. Security Considerations 11
5. References 11
5.1. Normative References 11
5.2. Informative References 12
Contributors 12
Authors' Addresses 12

1. Introduction

In some cases, it is desirable to use PAKE-based post-handshake authentication over TLS channel to execute password authentication between a client and a server, because this does not need to change the current TLS 1.3 [RFC8446] protocol stack and can defend against password leakages caused by a potential man-in-the-middle (MITM) attack on the underlying TLS channel. This strategy is often called defense-in-depth, the security of application-layer password authentication is still guaranteed even if the security of the underlying TLS-layer is broken. Optionally, this post-handshake authentication can be binded to the underlying TLS channel in order to strength password authentication, where the PAKE-based authentication will fail if the underlying TLS channel is broken. In addition, this post-handshake authentication is able to hide the client's identity from the network if the underlying TLS channel is secure.

Note that the post-handshake authentication via OPAQUE has been discussed in [I-D.draft-sullivan-tls-opaque-01], which utilizes the mechanism of Exported Authenticators in TLS 1.3 [RFC9261]. However, this mechanism is only applicable to these PAKE protocols, such as OPAQUE [I-D.draft-irtf-cfrg-opaque-18], where both the client and server own their long-term secret/public keys. This document provides a mechanism that uses PAKE as a post-handshake authentication for TLS 1.3 (called PAKE-based PHA for TLS 1.3) to achieve application-layer password authentication, which does not require two parties of PAKE protocols to possess long-term key pairs and supports PAKE algorithms negotiation and optional channel binding.

2. Terminology

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

This document uses terminology such as client, server, handshake and peer that are defined in Section 1.1 of [RFC8446].

3. Post-Handshake Authentication via PAKE for TLS 1.3

This section describes details of PAKE-based post-handshake authentication for TLS 1.3. In this document, four PAKE algorithms are considered: CPace [I-D.draft-irtf-cfrg-cpace-13], SPAKE2 [RFC9382], OPAQUE [I-D.draft-irtf-cfrg-opaque-18] and SPAKE2+ [RFC9383]. The former two are symmetric PAKE algorithms and the latter two are asymmetric PAKE algorithms.

3.1. PAKE Handshake Messages

To use PAKE as an application-layer password authentication over TLS 1.3 [RFC8446], we define PAKEHandshake messages which are used to negotiate PAKE algorithms and key exchange parameters and to complete PAKE-based authentication. The expected order of PAKE handshake messages is: PAKEClientHello, PAKEHelloRetryRequest, PAKEClientHello, PAKEServerHello, server PAKEFinished, client PAKEFinished.

```
enum {
    pake_client_hello (0),
    pake_server_hello (1),
    pake_hello_retry_request (2),
    pake_finished (3),
    pake_status (4),
    pake_message_hash (254),
    (255)
} PAKEHandshakeType;

struct {
    PAKEHandshakeType msg_type;           /* PAKE message type */
    uint24 length;                        /* bytes in message */
    select (PAKEHandshake.msg_type) {
        case pake_client_hello:           PAKEClientHello;
        case pake_server_hello:           PAKEServerHello;
        case pake_hello_retry_request:    PAKEHelloRetryRequest;
        case pake_finished:               PAKEFinished;
        case pake_status:                 PAKEStatus;
  };
} PAKEHandshake;
```

pake_client_hello: The PAKEClientHello message is used by the client to send its supported PAKE algorithm suites and PAKE shares for selected algorithm suites to the server.

pake_server_hello: The PAKEServerHello message is used by the server to send its PAKE share for the negotiated algorithm suite to the client.

pake_hello_retry_request: If the server does not support PAKE algorithm suites selected by the client in the PAKEClientHello message but supports other PAKE algorithm suites listed by the client, the server MUST use the PAKEHelloRetryRequest message to send a PAKE algorithm suite that is supported by both parties to the client.

pake_finished: The PAKEFinished message is used by the client or server to send a message authentication code (MAC) to its peer for identity authentication, key confirmation, and handshake integrity.

pake_status: The PAKEStatus message is used by the client or server to send the execution status of the PAKE protocol to its peer, indicating whether the PAKE protocol has been successfully executed.

pake_message_hash: When the server responds to a PAKEClientHello message with a PAKEHelloRetryRequest message, the value of the PAKEClientHello1 message is replaced with a specific synthetic handshake message of handshake type "pake_message_hash" containing Hash(PAKEClientHello1).

3.1.1. PAKE Client Hello

Structure of the PAKEClientHello message is defined as follows:

```
struct {
    PAKEAlgorithm supported_pake_algorithms<2..2^16-1>;
    opaque client_identity<0..2^16-1>;
    PAKEShareEntry client_shares<0..2^16-1>;
} PAKEClientHello;
```

* supported_pake_algorithms: A list of all PAKE algorithm suites supported by the client. The structure of PAKEAlgorithm is defined in Section 3.1.1 of [I-D.draft-guo-pake-in-tls-01], where the second bytes "0x00~0x7F" can be used to represent ciphersuites without channel binding and the second bytes "0x80~0xFF" can be used to represent ciphersuites with channel binding.

* client_identity: A client’s identity used in the PAKE algorithm.

* client_shares: A list of offered PAKEShareEntry values in descending order of client preference. The structure of PAKEShareEntry is defined in Section 3.1.2 of [I-D.draft-guo-pake-in-tls-01].

Note that the concatenation of the "random" value of the ClientHello message (see Section 4.1.2 of [RFC8446]) and the "random" value of the ServerHello message (see Section 4.1.3 of [RFC8446]) can be used as a unique session identifier sid of the CPace algorithm (see Section 3.1 of [I-D.draft-irtf-cfrg-cpace-13]).

3.1.2. PAKE Server Hello

Structure of the PAKEServerHello message is defined as follows:

```
struct {
    opaque server_identity<0..2^16-1>;
    PAKEShareEntry server_share;
} PAKEServerHello;
```

* server_identity: A server’s identity used in the PAKE algorithm.

* server_share: A single PAKEShareEntry value that is in the same PAKE algorithm suite as one of the client's shares. The structure of PAKEShareEntry is defined in Section 3.1.2 of [I-D.draft-guo-pake-in-tls-01].

3.1.3. PAKE Hello Retry Request

Structure of the PAKEHelloRetryRequest message is defined as follows:

```
struct {
    PAKEAlgorithm selected_pake_algorithm;
} PAKEHelloRetryRequest;
```

* selected_pake_algorithm: A PAKE algorithm suite selected by the server to correct mismatch algorithm suites with the client.

3.1.4. PAKE Finished

Structure of the PAKEFinished message is defined as follows:

```
struct {
    opaque pake_verify_data[MAC.length];
} PAKEFinished;
```

* pake_verify_data: A MAC value calculated by the client or server to provide to its peer for identity authentication, key confirmation and handshake integrity, and more details about this calculation will be given in Section 3.3. MAC is the MAC function negotiated in the PAKE algorithm suite, and MAC.length is its output length in bytes.

3.1.5. PAKE Status

Structure of the PAKEStatus message is defined as follows:

```
enum {
    pake_success_notify (0)
    pake_unexpected_message (1),
    pake_handshake_failure (2),
    pake_illegal_parameter (3),
    pake_decode_error (4),
    pake_decrypt_error (5),
    pake_insufficient_security (6),
    pake_internal_error (7),
    (255)
} PAKEStatusDescription;
```

```
struct {
    PAKEStatusDescription description;
} PAKEStatus;
```

pake_success_notify: This status notifies the client of the successful validation of its PAKEFinished message.

pake_unexpected_message: An inappropriate message (e.g., the wrong PAKE handshake message, etc.) was received. A peer which receives a PAKE handshake message in an unexpected order MUST abort the handshake with an "pake_unexpected_message" alert. This alert should never be observed in communication between proper implementations.

pake_handshake_failure: Receipt of a "pake_handshake_failure" alert message indicates that the sender was unable to negotiate an acceptable PAKE algorithm suite given the options available.

pake_illegal_parameter: A field in the PAKE handshake was incorrect or inconsistent with other fields. This alert is used for errors which conform to the formal protocol syntax but are otherwise incorrect.

pake_decode_error: A message could not be decoded because some field was out of the specified range or the length of the message was incorrect. This alert is used for errors where the message does not conform to the formal protocol syntax. This alert should never be observed in communication between proper implementations, except when messages were corrupted in the network.

pake_decrypt_error: A PAKE handshake cryptographic operation failed, including being unable to correctly verify a PAKEFinished message.

pake_insufficient_security: Returned instead of "pake_handshake_failure" when a negotiation has failed specifically because the server requires PAKE parameters more secure than those supported by the client.

pake_internal_error: An internal error unrelated to the peer or the correctness of the protocol (such as a memory allocation failure) makes it impossible to continue.

3.2. Channel Binding

This document defines two types for channel binding, which can be distinguished by the second byte of PAKE algorithm suites (see Section 3.1.1). The meaning of the types and the corresponding channel_binding_secret are described as follows:

* The "without_channel_binding" type means that no channel binding is used in the PAKE-based authentication, then the channel_binding_secret is a 0-byte value.

* The "with_channel_binding" type means that a channel binding is used in the PAKE-based authentication, then the channel_binding_secret is a 32-byte value. The computation of this secret is defined in Section 7.5 of [RFC8446] for TLS 1.3, and the corresponding inputs are defined in Section 2 of [RFC9266].

Assume that a TLS channel has been established between a client and a server, from which both parties can derive a unique secret that we call a channel binding secret in this document. According to the channel binding type of "tls-exporter" defined in [RFC9266], the channel binding secret is computed as follows.

channel_binding_secret = HKDF-Expand-Label(Derive-Secret(Secret, label, ""), "exporter", Hash(context_value), key_length)

Where Secret is the "exporter_master_secret" value in TLS 1.3 key schedule (see Section 7.1 of [RFC8446]), the label is the ASCII string "EXPORTER-Channel-Binding", the context_value is a zero-length string, the key_length is 32 bytes. The functions HKDF-Expand-Label and Derive-Secret were defined in Section 7.1 of [RFC8446].

3.3. PAKE-based Post-Handshake Authentication

After the TLS channel is established, if there is no MITM attack, the client and server can derive a same channel binding secret. If the channel binding is selected, then this secret will be used as another input to PAKE algorithms except for the password and thus take effect on the PAKE authentication process. If there is a MITM attack during the TLS channel establishment, that is, the client establishes a TLS channel A with the MITM attacker and the MITM attacker establishes a TLS channel B with the server, then the client and server will derive two different secrets respectively. Therefore, when the client and server execute the PAKE protocol with these inconsistent secrets, both parties can not pass the PAKE authentication successfully.

The protocol of PAKE-based post-handshake authentication for TLS 1.3 is described as follows.

(1) When PAKE-based post-handshake authentication for TLS 1.3 needs to be performed, it is REQUIRED to send the PAKEClientHello as a first PAKE handshake message. The client sets the "supported_pake_algorithms" field to a list of its supported PAKE algorithm suites, sets the "client_identity" field to its identity used to authenticate, and constructs the "client_shares" field by

selecting its perferred PAKE algorithm suites and computing their corresponding PAKE shares. The computation of PAKE shares SHOULD conform to the specification of the selected PAKE algorithms. Similar to TLS 1.3, the client MAY provide a single share or multiple shares in the "client_shares" field. The client then sends the PAKEClientHello message to the server.

(2) After receiving the PAKEClientHello message, the server first parses it to obtain "supported_pake_algorithms", "client_identity" and "client_shares" fields, uses the "client_identity" value to search a match password or password file, and negotiates a PAKE algorithm suite based on the "pake_algorithm" values included in the "client_shares" field. The server then sets the "server_identity" field to its identity (e.g., its host name), and constructs the "server_share" field by setting the negotiated PAKE algorithm suite and computing its corresponding PAKE share. Based on the received client’s share and its own secret, the server first derives a PAKE shared secret, and then derives a session key and a MAC key as follows.

session_key || mac_key = KDF(pake_shared_secret || channel_binding_secret, Hash(PAKEClientHello || PAKEServerHello), "TLS13PostHandshakeAuthPAKE", L)

Here, KDF(ikm, salt, info, L) is the key-derivation function (KDF) negotiated in the PAKE algorithm suite and the derived key length L relies on the underlying encryption function and MAC function; the PAKE shared secrets for different PAKEs are defined as follows:

* For CPace, this secret indicates the value ISK in Section 6.2 of [I-D.draft-irtf-cfrg-cpace-13].

* For SPAKE2, this secret indicates the concatenated value Ke || Ka in Section 3.3 of [RFC9382].

* For OPAQUE, this secret indicates the concatenated value Km2 || Km3 || session_key in Section 6.4.3 and Section 6.4.4 of [I-D.draft-irtf-cfrg-opaque-18].

* For SPAKE2+, this secret indicates the value K_main in Section 3.3 of [RFC9383].

Finally, the server computes its "pake_verify_data" value as follows, and sends PAKEServerHello and server PAKEFinished messages to the client.

server_pake_verify_data = MAC(mac_key, Hash(PAKEClientHello || PAKEServerHello))

(3) After receiving the PAKEServerHello and server PAKEFinished messages, the client first parses the former to obtain the "server_identity" and "server_share" fields, and parses the later to obtain the "pake_verify_data" field. Based on the received server's share and its own secret, the client derives a PAKE shared secret, and then derives a session key and a MAC key using the same way as the server side. The client then authenticates the server by verifying the server "pake_verify_data" value. If this verification succeeds, the client computes its "pake_verify_data" value as follows, and sends the client PAKEFinished message to the server.

client_pake_verify_data = MAC(mac_key, Hash(PAKEClientHello || PAKEServerHello || server PAKEFinished))

Otherwise, the client sends a PAKEStatus message with a "pake_decrypt_error" alert to the server.

(4) After receiving the client PAKEFinished message, the server first parses it to obtain the "pake_verify_data" field, then authenticates the client by verifying the client "pake_verify_data" value. If this verification succeeds, the server sends a PAKEStatus message with a "pake_success_notify" status to the client, otherwise sends a PAKEStatus message with a "pake_decrypt_error" alert to the client.

If the client has not provided a sufficient "client_shares" field (e.g., it includes only PAKE algorithm suites unacceptable to or unsupported by the server) in the first PAKEClientHello message, the server corrects the mismatch with a PAKEHelloRetryRequest message which contains a "selected_pake_algorithm" field, and the client needs to restart the handshake with a second PAKEClientHello message which MUST contain an appropriate "client_shares" field. In this case, the computation method of "pake_verify_data" values is changed as follows, where the "pake_message_hash" value represents 1 byte 0xFE as defined in Section 3.1, and Hash.length is the output length of the negotiated hash function in bytes.

server_pake_verify_data = MAC(mac_key, Hash(pake_message_hash || 00 00 Hash.length || Hash(PAKEClientHello1) || PAKEHelloRetryRequest || PAKEClientHello2 || PAKEServerHello))

client_pake_verify_data = MAC(mac_key, Hash(pake_message_hash || 00 00 Hash.length || Hash(PAKEClientHello1) || PAKEHelloRetryRequest || PAKEClientHello2 || PAKEServerHello || server PAKEFinished))

If no common PAKE parameters can be negotiated, the server MUST abort the handshake with either a "pake_handshake_failure" or "pake_insufficient_security" alert.

4. Security Considerations

This document describes how to execute PAKE-based post-handshake authentication for TLS 1.3. This execution deviates from the original PAKE protocols in two important ways: 1) the original keys of PAKEs are replaced with the session_key and mac_key in Section 3.3; 2) the key confirmation messages required in PAKEs are replaced with the PAKEFinished messages in Section 3.3. The former is because the session_key and mac_key are derived from a transcript, which includes the parameters required for PAKEs' key derivation; the latter is because the PAKEFinished messages compute a MAC over a transcript, which is a superset of the transcript required for PAKEs' key confirmation.

5. References

5.1. Normative References

[RFC9383] Taubert, T. and C. A. Wood, "SPAKE2+, an Augmented Password-Authenticated Key Exchange (PAKE) Protocol", RFC 9383, DOI 10.17487/RFC9383, September 2023, <https://www.rfc-editor.org/rfc/rfc9383>.

[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>.

[RFC9261] Sullivan, N., "Exported Authenticators in TLS", RFC 9261, DOI 10.17487/RFC9261, July 2022, <https://www.rfc-editor.org/rfc/rfc9261>.

[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>.

[RFC9382] Ladd, W., "SPAKE2, a Password-Authenticated Key Exchange", RFC 9382, DOI 10.17487/RFC9382, September 2023, <https://www.rfc-editor.org/rfc/rfc9382>.

[RFC9266] Whited, S., "Channel Bindings for TLS 1.3", RFC 9266, DOI 10.17487/RFC9266, July 2022, <https://www.rfc-editor.org/rfc/rfc9266>.

5.2. Informative References

[I-D.draft-irtf-cfrg-opaque-18]
Bourdrez, D., Krawczyk, H., Lewi, K., and C. A. Wood, "The OPAQUE Augmented PAKE Protocol", Work in Progress, Internet-Draft, draft-irtf-cfrg-opaque-18, 21 November 2024, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-opaque-18>.

[I-D.draft-sullivan-tls-opaque-01]
Sullivan, N., Krawczyk, H., Friel, O., and R. Barnes, "OPAQUE with TLS 1.3", Work in Progress, Internet-Draft, draft-sullivan-tls-opaque-01, 22 February 2021, <https://datatracker.ietf.org/doc/html/draft-sullivan-tls-opaque-01>.

[I-D.draft-irtf-cfrg-cpace-13]
Abdalla, M., Haase, B., and J. Hesse, "CPace, a balanced composable PAKE", Work in Progress, Internet-Draft, draft-irtf-cfrg-cpace-13, 14 October 2024, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-cpace-13>.

[I-D.draft-guo-pake-in-tls-01]
Guo, W., Xia, L., and J. Li, "PAKE Usage in TLS 1.3", Work in Progress, Internet-Draft, draft-guo-pake-in-tls-01, 24 February 2025, <https://datatracker.ietf.org/doc/html/draft-guo-pake-in-tls-01>.

Contributors

Yong Li
Huawei Technologies
Yong.Li1@huawei.com

Hui Ye
Huawei Technologies
yehui.ustc@huawei.com

Feng Geng
Huawei Technologies
gengfeng@huawei.com

Authors' Addresses

Wei Guo
Huawei Technologies
Email: guowei90@huawei.com

Liang Xia
Huawei Technologies
Email: frank.xialiang@huawei.com

Ji Li
Huawei Technologies
Email: liji100@huawei.com