

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 29 August 2022

C. Bartle
Apple, Inc.
N. Aviram
25 February 2022

Deprecating Obsolete Key Exchange Methods in TLS
draft-aviram-tls-deprecate-obsolete-kex-01

Abstract

This document makes several prescriptions regarding the following key exchange methods in TLS, most of which have been superseded by better options:

1. This document deprecates the use of RSA key exchange in TLS.
2. It limits the use of Diffie Hellman key exchange over a finite field to avoid known vulnerabilities and improper security properties.
3. It discourages the use of static elliptic curve Diffie Hellman cipher suites.

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 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	2
1.1. Requirements	4
2. Non-Ephemeral Diffie Hellman	4
3. Ephemeral Finite Field Diffie Hellman	4
4. RSA	5
5. IANA Considerations	5
6. Security Considerations	5
7. Acknowledgments	6
8. References	6
8.1. Normative References	6
8.2. Informative References	8
Appendix A. DH Cipher Suites Deprecated by This Document	10
Appendix B. ECDH Cipher Suites Whose Use Is Discouraged by This Document	13
Appendix C. DHE Cipher Suites Referred to by This Document	15
Appendix D. RSA Cipher Suites Deprecated by This Document	18
Authors' Addresses	20

1. Introduction

TLS supports a variety of key exchange algorithms, including RSA, Diffie Hellman over a finite field, and elliptic curve Diffie Hellman (ECDH).

Diffie Hellman key exchange, over any group, comes in ephemeral and non-ephemeral varieties. Non-ephemeral DH algorithms use static DH public keys included in the authenticating peer's certificate; see [RFC4492] for discussion. In contrast, ephemeral DH algorithms use ephemeral DH public keys sent in the handshake and authenticated by the peer's certificate. Ephemeral and non-ephemeral finite field DH algorithms are called DHE and DH (or FFDHE and FFDH), respectively, and ephemeral and non-ephemeral elliptic curve DH algorithms are called ECDHE and ECDH, respectively [RFC4492].

In general, non-ephemeral cipher suites are not recommended due to their lack of forward secrecy. However, as demonstrated by the [Raccoon] attack on finite-field DH, public key reuse, either via

non-ephemeral cipher suites or reused keys with ephemeral cipher suites, can lead to timing side channels that may leak connection secrets. For elliptic curve DH, invalid curve attacks similarly exploit secret reuse in order to break security [ICA], further demonstrating the risk of reusing public keys. While both side channels can be avoided in implementations, experience shows that in practice, implementations may fail to thwart such attacks due to the complexity and number of the required mitigations.

Additionally, RSA key exchange suffers from security problems that are independent of implementation choices as well as problems that stem purely from the difficulty of implementing security countermeasures correctly.

At a rough glance, the problems affecting FFDHE are as follows:

1. FFDHE suffers from interoperability problems because there is no mechanism for negotiating the group size, and some implementations only support small group sizes (see [RFC7919], Section 1).
2. In practice, some operators use 1024-bit FFDHE groups since this is the maximum size that ensures wide support (see [RFC7919], Section 1). This size leaves only a small security margin vs. the current discrete log record, which stands at 795 bits [DLOG795].
3. Expanding on the previous point, just a handful of very large computations allow an attacker to cheaply decrypt a relatively large fraction of FFDHE traffic (namely, traffic encrypted using particular standardized groups) [weak-dh].
4. When secrets are not fully ephemeral, FFDHE suffers from the [Raccoon] side channel attack. (Note that FFDH is inherently vulnerable to the Raccoon attack unless constant-time mitigations are employed.)
5. FFDHE groups may have small subgroups, which enables several attacks [subgroups].

The problems affecting RSA key exchange are as follows:

1. RSA key exchange offers no forward secrecy, by construction.
2. RSA key exchange may be vulnerable to Bleichenbacher's attack [BLEI]. Experience shows that variants of this attack arise every few years because implementing the relevant countermeasure correctly is difficult (see [ROBOT], [NEW-BLEI], [DROWN]).

3. In addition to the above point, there is no convenient mechanism in TLS for the domain separation of keys. Therefore, a single endpoint that is vulnerable to Bleichenbacher's attack would affect all endpoints sharing the same RSA key (see [XPROT], [DROWN]).

Given these problems, this document updates [RFC4346], [RFC5246], [RFC4162], [RFC6347], [RFC5932], [RFC5288], [RFC6209], [RFC6367], [RFC8422], [RFC5289], and [RFC5469] to deprecate cipher suites with key reuse.

1.1. Requirements

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. Non-Ephemeral Diffie Hellman

Clients MUST NOT offer non-ephemeral FFDH cipher suites in TLS 1.2 connections. (Note that TLS 1.0 and 1.1 are deprecated by [RFC8996] and TLS 1.3 does not support FFDH [RFC8446].) This includes all cipher suites listed in the table in Appendix A.

Clients SHOULD NOT offer non-ephemeral ECDH cipher suites in TLS 1.2 connections. (Note that TLS 1.0 and 1.1 are deprecated by [RFC8996] and TLS 1.3 does not support ECDH [RFC8446].) This includes all cipher suites listed in the table in Appendix B.

3. Ephemeral Finite Field Diffie Hellman

Clients and servers MAY offer fully ephemeral FFDHE cipher suites in TLS 1.2 connections under the following conditions:

1. Clients and servers MUST NOT reuse ephemeral DHE public keys across TLS connections for all existing (and future) TLS versions. Doing so invalidates forward secrecy properties of these connections. For DHE, such reuse may also lead to vulnerabilities such as those used in the [Raccoon] attack. See Section 6 for related discussion.
2. The group is one of the following well-known groups described in [RFC7919]: ffdhe2048, ffdhe3072, ffdhe4096, ffdhe6144, ffdhe8192.

(Note that TLS 1.0 and 1.1 are deprecated by [RFC8996]. TLS 1.3 satisfies the second point above [RFC8446] and is not vulnerable to the [Raccoon] Attack.)

We note that, previously, supporting the broadest range of clients would have required supporting either RSA key exchange or 1024-bit FFDHE. This is no longer the case, and it is possible to support most clients released since circa 2015 using 2048-bit FFDHE or more modern key exchange methods, and without RSA key exchange [server_side_tls].

All the cipher suites that do not meet the above requirements are listed in the table in Appendix C.

4. RSA

Clients and servers MUST NOT offer RSA cipher suites in TLS 1.2 connections. (Note that TLS 1.0 and 1.1 are deprecated by [RFC8996], and TLS 1.3 does not support static RSA [RFC8446].) This includes all cipher suites listed in the table in Appendix D. Note that these cipher suites are already marked as not recommended in the "TLS Cipher Suites" registry.

5. IANA Considerations

This document makes no requests to IANA. Note that all cipher suites listed in Section 4 and in Section 2 are already marked as not recommended in the "TLS Cipher Suites" registry.

6. Security Considerations

Non-ephemeral finite field DH cipher suites (TLS_DH_*), as well as ephemeral key reuse for finite field DH cipher suites, are prohibited due to the [Raccoon] attack. Both are already considered bad practice since they do not provide forward secrecy. However, Raccoon revealed that timing side channels in processing TLS premaster secrets may be exploited to reveal the encrypted premaster secret.

As for non-ephemeral elliptic curve DH cipher suites, forgoing forward secrecy not only allows retroactive decryption in the event of key compromise but may also enable a broad category of attacks where the attacker exploits key reuse to repeatedly query a cryptographic secret.

This category includes, but is not necessarily limited to, the following examples:

1. Invalid curve attacks, where the attacker exploits key reuse to repeatedly query and eventually learn the key itself. These attacks have been shown to be practical against real-world TLS implementations [ICA].
2. Side channel attacks, where the attacker exploits key reuse and an additional side channel to learn a cryptographic secret. As one example of such attacks, refer to [MAY4].
3. Fault attacks, where the attacker exploits key reuse and incorrect calculations to learn a cryptographic secret. As one example of such attacks, see [PARIS256].

Such attacks are often implementation-dependent, including the above examples. However, these examples demonstrate that building a system that reuses keys and avoids this category of attacks is difficult in practice. In contrast, avoiding key reuse not only prevents decryption in the event of key compromise, but also precludes this category of attacks altogether. Therefore, this document discourages the reuse of elliptic curve DH public keys.

7. Acknowledgments

This document was inspired by discussions on the TLS WG mailing list and a suggestion by Filippo Valsorda following the release of the [Raccoon] attack. Thanks to Christopher A. Wood for writing up the initial draft of this document.

8. References

8.1. 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/info/rfc2119>>.
- [RFC4162] Lee, H.J., Yoon, J.H., and J.I. Lee, "Addition of SEED Cipher Suites to Transport Layer Security (TLS)", RFC 4162, DOI 10.17487/RFC4162, August 2005, <<https://www.rfc-editor.org/info/rfc4162>>.
- [RFC4279] Eronen, P., Ed. and H. Tschofenig, Ed., "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)", RFC 4279, DOI 10.17487/RFC4279, December 2005, <<https://www.rfc-editor.org/info/rfc4279>>.

- [RFC4346] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1", RFC 4346, DOI 10.17487/RFC4346, April 2006, <<https://www.rfc-editor.org/info/rfc4346>>.
- [RFC4785] Blumenthal, U. and P. Goel, "Pre-Shared Key (PSK) Ciphersuites with NULL Encryption for Transport Layer Security (TLS)", RFC 4785, DOI 10.17487/RFC4785, January 2007, <<https://www.rfc-editor.org/info/rfc4785>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<https://www.rfc-editor.org/info/rfc5246>>.
- [RFC5288] Salowey, J., Choudhury, A., and D. McGrew, "AES Galois Counter Mode (GCM) Cipher Suites for TLS", RFC 5288, DOI 10.17487/RFC5288, August 2008, <<https://www.rfc-editor.org/info/rfc5288>>.
- [RFC5289] Rescorla, E., "TLS Elliptic Curve Cipher Suites with SHA-256/384 and AES Galois Counter Mode (GCM)", RFC 5289, DOI 10.17487/RFC5289, August 2008, <<https://www.rfc-editor.org/info/rfc5289>>.
- [RFC5469] Eronen, P., Ed., "DES and IDEA Cipher Suites for Transport Layer Security (TLS)", RFC 5469, DOI 10.17487/RFC5469, February 2009, <<https://www.rfc-editor.org/info/rfc5469>>.
- [RFC5487] Badra, M., "Pre-Shared Key Cipher Suites for TLS with SHA-256/384 and AES Galois Counter Mode", RFC 5487, DOI 10.17487/RFC5487, March 2009, <<https://www.rfc-editor.org/info/rfc5487>>.
- [RFC5932] Kato, A., Kanda, M., and S. Kanno, "Camellia Cipher Suites for TLS", RFC 5932, DOI 10.17487/RFC5932, June 2010, <<https://www.rfc-editor.org/info/rfc5932>>.
- [RFC6209] Kim, W., Lee, J., Park, J., and D. Kwon, "Addition of the ARIA Cipher Suites to Transport Layer Security (TLS)", RFC 6209, DOI 10.17487/RFC6209, April 2011, <<https://www.rfc-editor.org/info/rfc6209>>.
- [RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <<https://www.rfc-editor.org/info/rfc6347>>.

- [RFC6367] Kanno, S. and M. Kanda, "Addition of the Camellia Cipher Suites to Transport Layer Security (TLS)", RFC 6367, DOI 10.17487/RFC6367, September 2011, <<https://www.rfc-editor.org/info/rfc6367>>.
- [RFC6655] McGrew, D. and D. Bailey, "AES-CCM Cipher Suites for Transport Layer Security (TLS)", RFC 6655, DOI 10.17487/RFC6655, July 2012, <<https://www.rfc-editor.org/info/rfc6655>>.
- [RFC7905] Langley, A., Chang, W., Mavrogiannopoulos, N., Strombergson, J., and S. Josefsson, "ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)", RFC 7905, DOI 10.17487/RFC7905, June 2016, <<https://www.rfc-editor.org/info/rfc7905>>.
- [RFC7919] Gillmor, D., "Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)", RFC 7919, DOI 10.17487/RFC7919, August 2016, <<https://www.rfc-editor.org/info/rfc7919>>.
- [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/info/rfc8174>>.
- [RFC8422] Nir, Y., Josefsson, S., and M. Pegourie-Gonnard, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier", RFC 8422, DOI 10.17487/RFC8422, August 2018, <<https://www.rfc-editor.org/info/rfc8422>>.
- [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/info/rfc8446>>.
- [RFC8996] Moriarty, K. and S. Farrell, "Deprecating TLS 1.0 and TLS 1.1", BCP 195, RFC 8996, DOI 10.17487/RFC8996, March 2021, <<https://www.rfc-editor.org/info/rfc8996>>.

8.2. Informative References

- [BLEI] Bleichenbacher, D., "Chosen Ciphertext Attacks against Protocols Based on RSA Encryption Standard PKCS #1", Advances in Cryptology -- CRYPTO'98, LNCS vol. 1462, pages: 1-12 , 1998.

- [DLOG795] Boudot, F., Gaudry, P., Guillevis, A., Heninger, N., Thomé, E., and P. Zimmermann, "Comparing the difficulty of factorization and discrete logarithm: a 240-digit experiment", 17 August 2020, <<https://eprint.iacr.org/2020/697>>.
- [DROWN] Aviram, N., Schinzel, S., Somorovsky, J., Heninger, N., Dankel, M., Steube, J., Valenta, L., Adrian, D., Halderman, J. A., Dukhovni, V., Käsper, E., Cohney, S., Engels, S., Paar, C., and Y. Shavitt, "DROWN: Breaking TLS using SSLv2", August 2016, <<https://drownattack.com/drown-attack-paper.pdf>>.
- [ICA] Jager, T., Schwenk, J., and J. Somorovsky, "Practical invalid curve attacks on TLS-ECDH", 21 September 2015, <<https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.704.7932&rep=rep1&type=pdf>>.
- [MAY4] Genkin, D., Valenta, L., and Y. Yarom, "May the fourth be with you: A microarchitectural side channel attack on several real-world applications of curve25519", n.d., <<https://dl.acm.org/doi/pdf/10.1145/3133956.3134029>>.
- [NEW-BLEI] Meyer, C., Somorovsky, J., Weiss, E., Schwenk, J., Schinzel, S., and E. Tews, "Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks", August 2014, <<https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-meyer.pdf>>.
- [PARIS256] Devlin, S. and F. Valsorda, "The PARIS256 Attack", n.d., <<https://i.blackhat.com/us-18/Wed-August-8/us-18-Valsorda-Squeezing-A-Key-Through-A-Carry-Bit-wp.pdf>>.
- [Raccoon] Merget, R., Brinkmann, M., Aviram, N., Somorovsky, J., Mittmann, J., and J. Schwenk, "Raccoon Attack: Finding and Exploiting Most-Significant-Bit-Oracles in TLS-DH(E)", 9 September 2020, <<https://raccoon-attack.com/RaccoonAttack.pdf>>.
- [RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", RFC 4492, DOI 10.17487/RFC4492, May 2006, <<https://www.rfc-editor.org/info/rfc4492>>.

- [ROBOT] Boeck, H., Somorovsky, J., and C. Young, "Return Of Bleichenbacher's Oracle Threat (ROBOT)", 27th USENIX Security Symposium , 2018.
- [SC-tls-des-idea-ciphers-to-historic]
"Moving single-DES and IDEA TLS ciphersuites to Historic", 25 January 2021, <<https://datatracker.ietf.org/doc/status-change-tls-des-idea-ciphers-to-historic/>>.
- [server_side_tls]
King, A., "Server Side TLS", July 2020, <https://wiki.mozilla.org/Security/Server_Side_TLS>.
- [subgroups]
Valenta, L., Adrian, D., Sanso, A., Cohnsey, S., Fried, J., Hastings, M., Halderman, J. A., and N. Heninger, "Measuring small subgroup attacks against Diffie-Hellman", 15 October 2016, <<https://eprint.iacr.org/2016/995/20161017:193515>>.
- [weak-dh] Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J. A., Heninger, N., Springall, D., Thomé, E., Valenta, L., VanderSloot, B., Wustrow, E., Zanella-Béguélin, S., and P. Zimmermann, "Weak Diffie-Hellman and the Logjam Attack", October 2015, <<https://weakdh.org/>>.
- [XPROT] Jager, T., Schwenk, J., and J. Somorovsky, "On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption", Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security , 2015.

Appendix A. DH Cipher Suites Deprecated by This Document

Ciphersuite	Reference
TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	[RFC4346]
TLS_DH_DSS_WITH_DES_CBC_SHA	[RFC5469]
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA	[RFC5246]
TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	[RFC4346]
TLS_DH_RSA_WITH_DES_CBC_SHA	[RFC5469]
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA	[RFC5246]

TLS_DH_anon_EXPORT_WITH_RC4_40_MD5	[RFC4346] [RFC6347]
TLS_DH_anon_WITH_RC4_128_MD5	[RFC5246] [RFC6347]
TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA	[RFC4346]
TLS_DH_anon_WITH_DES_CBC_SHA	[RFC5469]
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA	[RFC5246]
TLS_DH_DSS_WITH_AES_128_CBC_SHA	[RFC5246]
TLS_DH_RSA_WITH_AES_128_CBC_SHA	[RFC5246]
TLS_DH_anon_WITH_AES_128_CBC_SHA	[RFC5246]
TLS_DH_DSS_WITH_AES_256_CBC_SHA	[RFC5246]
TLS_DH_RSA_WITH_AES_256_CBC_SHA	[RFC5246]
TLS_DH_anon_WITH_AES_256_CBC_SHA	[RFC5246]
TLS_DH_DSS_WITH_AES_128_CBC_SHA256	[RFC5246]
TLS_DH_RSA_WITH_AES_128_CBC_SHA256	[RFC5246]
TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA	[RFC5932]
TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA	[RFC5932]
TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA	[RFC5932]
TLS_DH_DSS_WITH_AES_256_CBC_SHA256	[RFC5246]
TLS_DH_RSA_WITH_AES_256_CBC_SHA256	[RFC5246]
TLS_DH_anon_WITH_AES_128_CBC_SHA256	[RFC5246]
TLS_DH_anon_WITH_AES_256_CBC_SHA256	[RFC5246]
TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA	[RFC5932]
TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA	[RFC5932]
TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA	[RFC5932]
TLS_DH_DSS_WITH_SEED_CBC_SHA	[RFC4162]

TLS_DH_RSA_WITH_SEED_CBC_SHA	[RFC4162]
TLS_DH_anon_WITH_SEED_CBC_SHA	[RFC4162]
TLS_DH_RSA_WITH_AES_128_GCM_SHA256	[RFC5288]
TLS_DH_RSA_WITH_AES_256_GCM_SHA384	[RFC5288]
TLS_DH_DSS_WITH_AES_128_GCM_SHA256	[RFC5288]
TLS_DH_DSS_WITH_AES_256_GCM_SHA384	[RFC5288]
TLS_DH_anon_WITH_AES_128_GCM_SHA256	[RFC5288]
TLS_DH_anon_WITH_AES_256_GCM_SHA384	[RFC5288]
TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA256	[RFC5932]
TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA256	[RFC5932]
TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA256	[RFC5932]
TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA256	[RFC5932]
TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA256	[RFC5932]
TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA256	[RFC5932]
TLS_DH_DSS_WITH_ARIA_128_CBC_SHA256	[RFC6209]
TLS_DH_DSS_WITH_ARIA_256_CBC_SHA384	[RFC6209]
TLS_DH_RSA_WITH_ARIA_128_CBC_SHA256	[RFC6209]
TLS_DH_RSA_WITH_ARIA_256_CBC_SHA384	[RFC6209]
TLS_DH_anon_WITH_ARIA_128_CBC_SHA256	[RFC6209]
TLS_DH_anon_WITH_ARIA_256_CBC_SHA384	[RFC6209]
TLS_DH_RSA_WITH_ARIA_128_GCM_SHA256	[RFC6209]
TLS_DH_RSA_WITH_ARIA_256_GCM_SHA384	[RFC6209]
TLS_DH_DSS_WITH_ARIA_128_GCM_SHA256	[RFC6209]
TLS_DH_DSS_WITH_ARIA_256_GCM_SHA384	[RFC6209]

TLS_DH_anon_WITH_ARIA_128_GCM_SHA256	[RFC6209]
TLS_DH_anon_WITH_ARIA_256_GCM_SHA384	[RFC6209]
TLS_DH_RSA_WITH_CAMELLIA_128_GCM_SHA256	[RFC6367]
TLS_DH_RSA_WITH_CAMELLIA_256_GCM_SHA384	[RFC6367]
TLS_DH_DSS_WITH_CAMELLIA_128_GCM_SHA256	[RFC6367]
TLS_DH_DSS_WITH_CAMELLIA_256_GCM_SHA384	[RFC6367]
TLS_DH_anon_WITH_CAMELLIA_128_GCM_SHA256	[RFC6367]
TLS_DH_anon_WITH_CAMELLIA_256_GCM_SHA384	[RFC6367]

Table 1

Appendix B. ECDH Cipher Suites Whose Use Is Discouraged by This Document

Ciphersuite	Reference
TLS_ECDH_ECDSA_WITH_NULL_SHA	[RFC8422]
TLS_ECDH_ECDSA_WITH_RC4_128_SHA	[RFC8422] [RFC6347]
TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA	[RFC8422]
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA	[RFC8422]
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA	[RFC8422]
TLS_ECDH_RSA_WITH_NULL_SHA	[RFC8422]
TLS_ECDH_RSA_WITH_RC4_128_SHA	[RFC8422] [RFC6347]
TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA	[RFC8422]
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA	[RFC8422]
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA	[RFC8422]
TLS_ECDH_anon_WITH_NULL_SHA	[RFC8422]

TLS_ECDH_anon_WITH_RC4_128_SHA	[RFC8422] [RFC6347]
TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA	[RFC8422]
TLS_ECDH_anon_WITH_AES_128_CBC_SHA	[RFC8422]
TLS_ECDH_anon_WITH_AES_256_CBC_SHA	[RFC8422]
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256	[RFC5289]
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384	[RFC5289]
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256	[RFC5289]
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384	[RFC5289]
TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256	[RFC5289]
TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384	[RFC5289]
TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256	[RFC5289]
TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384	[RFC5289]
TLS_ECDH_ECDSA_WITH_ARIA_128_CBC_SHA256	[RFC6209]
TLS_ECDH_ECDSA_WITH_ARIA_256_CBC_SHA384	[RFC6209]
TLS_ECDH_RSA_WITH_ARIA_128_CBC_SHA256	[RFC6209]
TLS_ECDH_RSA_WITH_ARIA_256_CBC_SHA384	[RFC6209]
TLS_ECDH_ECDSA_WITH_ARIA_128_GCM_SHA256	[RFC6209]
TLS_ECDH_ECDSA_WITH_ARIA_256_GCM_SHA384	[RFC6209]
TLS_ECDH_RSA_WITH_ARIA_128_GCM_SHA256	[RFC6209]
TLS_ECDH_RSA_WITH_ARIA_256_GCM_SHA384	[RFC6209]
TLS_ECDH_ECDSA_WITH_CAMELLIA_128_CBC_SHA256	[RFC6367]
TLS_ECDH_ECDSA_WITH_CAMELLIA_256_CBC_SHA384	[RFC6367]
TLS_ECDH_RSA_WITH_CAMELLIA_128_CBC_SHA256	[RFC6367]
TLS_ECDH_RSA_WITH_CAMELLIA_256_CBC_SHA384	[RFC6367]

TLS_ECDH_ECDSA_WITH_CAMELLIA_128_GCM_SHA256	[RFC6367]	
+-----+-----+-----+		
TLS_ECDH_ECDSA_WITH_CAMELLIA_256_GCM_SHA384	[RFC6367]	
+-----+-----+-----+		
TLS_ECDH_RSA_WITH_CAMELLIA_128_GCM_SHA256	[RFC6367]	
+-----+-----+-----+		
TLS_ECDH_RSA_WITH_CAMELLIA_256_GCM_SHA384	[RFC6367]	
+-----+-----+-----+		

Table 2

Appendix C. DHE Cipher Suites Referred to by This Document

=====+	
Ciphersuite	Reference
+-----+	
TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	[RFC4346]
+-----+	
TLS_DHE_DSS_WITH_DES_CBC_SHA	[RFC5469] [SC-tls-des-idea-ciphers-to-historic]
+-----+	
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	[RFC5246]
+-----+	
TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	[RFC4346]
+-----+	
TLS_DHE_RSA_WITH_DES_CBC_SHA	[RFC5469] [SC-tls-des-idea-ciphers-to-historic]
+-----+	
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	[RFC5246]
+-----+	
TLS_DHE_PSK_WITH_NULL_SHA	[RFC4785]
+-----+	
TLS_DHE_DSS_WITH_AES_128_CBC_SHA	[RFC5246]
+-----+	
TLS_DHE_RSA_WITH_AES_128_CBC_SHA	[RFC5246]
+-----+	
TLS_DHE_DSS_WITH_AES_256_CBC_SHA	[RFC5246]
+-----+	

-----+		
	TLS_DHE_RSA_WITH_AES_256_CBC_SHA	[RFC5246]
+-----+		
-----+		
	TLS_DHE_DSS_WITH_AES_128_CBC_SHA256	[RFC5246]
+-----+		
-----+		
	TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA	[RFC5932]
+-----+		
-----+		
	TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA	[RFC5932]
+-----+		
-----+		
	TLS_DHE_RSA_WITH_AES_128_CBC_SHA256	[RFC5246]
+-----+		
-----+		
	TLS_DHE_DSS_WITH_AES_256_CBC_SHA256	[RFC5246]
+-----+		
-----+		

TLS_DHE_RSA_WITH_AES_256_CBC_SHA256	[RFC5246]
+-----+	+-----+
TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA	[RFC5932]
+-----+	+-----+
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA	[RFC5932]
+-----+	+-----+
TLS_DHE_PSK_WITH_RC4_128_SHA	[RFC4279] [RFC6347]
+-----+	+-----+
TLS_DHE_PSK_WITH_3DES_EDE_CBC_SHA	[RFC4279]
+-----+	+-----+
TLS_DHE_PSK_WITH_AES_128_CBC_SHA	[RFC4279]
+-----+	+-----+
TLS_DHE_PSK_WITH_AES_256_CBC_SHA	[RFC4279]
+-----+	+-----+
TLS_DHE_DSS_WITH_SEED_CBC_SHA	[RFC4162]
+-----+	+-----+
TLS_DHE_RSA_WITH_SEED_CBC_SHA	[RFC4162]
+-----+	+-----+
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256	[RFC5288]
+-----+	+-----+
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	[RFC5288]
+-----+	+-----+
TLS_DHE_DSS_WITH_AES_128_GCM_SHA256	[RFC5288]
+-----+	+-----+
TLS_DHE_DSS_WITH_AES_256_GCM_SHA384	[RFC5288]
+-----+	+-----+
TLS_DHE_PSK_WITH_AES_128_GCM_SHA256	[RFC5487]
+-----+	+-----+
TLS_DHE_PSK_WITH_AES_256_GCM_SHA384	[RFC5487]
+-----+	+-----+

-----+-----	
-----+-----	
TLS_DHE_PSK_WITH_AES_128_CBC_SHA256	[RFC5487]
-----+-----	
-----+-----	
TLS_DHE_PSK_WITH_AES_256_CBC_SHA384	[RFC5487]
-----+-----	
-----+-----	
TLS_DHE_PSK_WITH_NULL_SHA256	[RFC5487]
-----+-----	
-----+-----	
TLS_DHE_PSK_WITH_NULL_SHA384	[RFC5487]
-----+-----	
-----+-----	
TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA256	[RFC5932]
-----+-----	
-----+-----	
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA256	[RFC5932]
-----+-----	
-----+-----	
TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA256	[RFC5932]
-----+-----	
-----+-----	
TLS_DHE_DSS_WITH_ARIA_128_CBC_SHA256	[RFC6209]
-----+-----	
-----+-----	

TLS_DHE_DSS_WITH_ARIA_256_CBC_SHA384	[RFC6209]
+	+
-----+	
TLS_DHE_RSA_WITH_ARIA_128_CBC_SHA256	[RFC6209]
+	+
-----+	
TLS_DHE_RSA_WITH_ARIA_256_CBC_SHA384	[RFC6209]
+	+
-----+	
TLS_DHE_RSA_WITH_ARIA_128_GCM_SHA256	[RFC6209]
+	+
-----+	
TLS_DHE_RSA_WITH_ARIA_256_GCM_SHA384	[RFC6209]
+	+
-----+	
TLS_DHE_DSS_WITH_ARIA_128_GCM_SHA256	[RFC6209]
+	+
-----+	
TLS_DHE_DSS_WITH_ARIA_256_GCM_SHA384	[RFC6209]
+	+
-----+	
TLS_DHE_PSK_WITH_ARIA_128_CBC_SHA256	[RFC6209]
+	+
-----+	
TLS_DHE_PSK_WITH_ARIA_256_CBC_SHA384	[RFC6209]
+	+
-----+	
TLS_DHE_PSK_WITH_ARIA_128_GCM_SHA256	[RFC6209]
+	+
-----+	
TLS_DHE_PSK_WITH_ARIA_256_GCM_SHA384	[RFC6209]
+	+
-----+	
TLS_DHE_RSA_WITH_CAMELLIA_128_GCM_SHA256	[RFC6367]
+	+
-----+	
TLS_DHE_RSA_WITH_CAMELLIA_256_GCM_SHA384	[RFC6367]
+	+
-----+	
TLS_DHE_DSS_WITH_CAMELLIA_128_GCM_SHA256	[RFC6367]
+	+
-----+	
TLS_DHE_DSS_WITH_CAMELLIA_256_GCM_SHA384	[RFC6367]
+	+
-----+	

-----+-----+-----	
-----+-----+-----	
TLS_DHE_PSK_WITH_CAMELLIA_128_GCM_SHA256	[RFC6367]
-----+-----+-----	
-----+-----+-----	
TLS_DHE_PSK_WITH_CAMELLIA_256_GCM_SHA384	[RFC6367]
-----+-----+-----	
-----+-----+-----	
TLS_DHE_PSK_WITH_CAMELLIA_128_CBC_SHA256	[RFC6367]
-----+-----+-----	
-----+-----+-----	
TLS_DHE_PSK_WITH_CAMELLIA_256_CBC_SHA384	[RFC6367]
-----+-----+-----	
-----+-----+-----	
TLS_DHE_RSA_WITH_AES_128_CCM	[RFC6655]
-----+-----+-----	
-----+-----+-----	
TLS_DHE_RSA_WITH_AES_256_CCM	[RFC6655]
-----+-----+-----	
-----+-----+-----	
TLS_DHE_RSA_WITH_AES_128_CCM_8	[RFC6655]
-----+-----+-----	
-----+-----+-----	
TLS_DHE_RSA_WITH_AES_256_CCM_8	[RFC6655]
-----+-----+-----	
-----+-----+-----	
TLS_DHE_PSK_WITH_AES_128_CCM	[RFC6655]
-----+-----+-----	
-----+-----+-----	

TLS_DHE_PSK_WITH_AES_256_CCM	[RFC6655]
+-----+	+-----+
TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256	[RFC7905]
+-----+	+-----+
TLS_DHE_PSK_WITH_CHACHA20_POLY1305_SHA256	[RFC7905]
+-----+	+-----+

Table 3

Appendix D. RSA Cipher Suites Deprecated by This Document

=====+ Ciphersuite +=====+	=====+ Reference +=====+
TLS_RSA_WITH_NULL_MD5 +-----+	[RFC5246] +-----+
TLS_RSA_WITH_NULL_SHA +-----+	[RFC5246] +-----+
TLS_RSA_EXPORT_WITH_RC4_40_MD5 +-----+	[RFC4346] [RFC6347] +-----+
TLS_RSA_WITH_RC4_128_MD5 +-----+	[RFC5246] [RFC6347] +-----+
TLS_RSA_WITH_RC4_128_SHA +-----+	[RFC5246] [RFC6347] +-----+
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5 +-----+	[RFC4346] +-----+
TLS_RSA_WITH_IDEA_CBC_SHA o-historic] +-----+	[RFC5469] [SC-tls-des-idea-ciphers-t o-historic] +-----+
TLS_RSA_EXPORT_WITH_DES40_CBC_SHA +-----+	[RFC4346] +-----+
TLS_RSA_WITH_DES_CBC_SHA o-historic] +-----+	[RFC5469] [SC-tls-des-idea-ciphers-t o-historic] +-----+

-----+ TLS_RSA_WITH_3DES_EDE_CBC_SHA +-----+-----+ -----+	[RFC5246]
-----+ TLS_RSA_PSK_WITH_NULL_SHA +-----+-----+ -----+	[RFC4785]
-----+ TLS_RSA_WITH_AES_128_CBC_SHA +-----+-----+ -----+	[RFC5246]
-----+ TLS_RSA_WITH_AES_256_CBC_SHA +-----+-----+ -----+	[RFC5246]
-----+ TLS_RSA_WITH_NULL_SHA256 +-----+-----+ -----+	[RFC5246]
-----+ TLS_RSA_WITH_AES_128_CBC_SHA256 +-----+-----+ -----+	[RFC5246]
-----+ TLS_RSA_WITH_AES_256_CBC_SHA256 +-----+-----+ -----+	[RFC5246]
-----+ TLS_RSA_WITH_CAMELLIA_128_CBC_SHA +-----+-----+ -----+	[RFC5932]

TLS_RSA_WITH_CAMELLIA_256_CBC_SHA	[RFC5932]
+-----+	+-----+
TLS_RSA_PSK_WITH_RC4_128_SHA	[RFC4279] [RFC6347]
+-----+	+-----+
TLS_RSA_PSK_WITH_3DES_EDE_CBC_SHA	[RFC4279]
+-----+	+-----+
TLS_RSA_PSK_WITH_AES_128_CBC_SHA	[RFC4279]
+-----+	+-----+
TLS_RSA_PSK_WITH_AES_256_CBC_SHA	[RFC4279]
+-----+	+-----+
TLS_RSA_WITH_SEED_CBC_SHA	[RFC4162]
+-----+	+-----+
TLS_RSA_WITH_AES_128_GCM_SHA256	[RFC5288]
+-----+	+-----+
TLS_RSA_WITH_AES_256_GCM_SHA384	[RFC5288]
+-----+	+-----+
TLS_RSA_PSK_WITH_AES_128_GCM_SHA256	[RFC5487]
+-----+	+-----+
TLS_RSA_PSK_WITH_AES_256_GCM_SHA384	[RFC5487]
+-----+	+-----+
TLS_RSA_PSK_WITH_AES_128_CBC_SHA256	[RFC5487]
+-----+	+-----+
TLS_RSA_PSK_WITH_AES_256_CBC_SHA384	[RFC5487]
+-----+	+-----+
TLS_RSA_PSK_WITH_NULL_SHA256	[RFC5487]
+-----+	+-----+
TLS_RSA_PSK_WITH_NULL_SHA384	[RFC5487]
+-----+	+-----+
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA256	[RFC5932]

-----+-----	-----+-----
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA256	[RFC5932]
-----+-----	-----+-----
TLS_RSA_WITH_ARIA_128_CBC_SHA256	[RFC6209]
-----+-----	-----+-----
TLS_RSA_WITH_ARIA_256_CBC_SHA384	[RFC6209]
-----+-----	-----+-----
TLS_RSA_WITH_ARIA_128_GCM_SHA256	[RFC6209]
-----+-----	-----+-----
TLS_RSA_WITH_ARIA_256_GCM_SHA384	[RFC6209]
-----+-----	-----+-----
TLS_RSA_PSK_WITH_ARIA_128_CBC_SHA256	[RFC6209]
-----+-----	-----+-----
TLS_RSA_PSK_WITH_ARIA_256_CBC_SHA384	[RFC6209]
-----+-----	-----+-----
TLS_RSA_PSK_WITH_ARIA_128_GCM_SHA256	[RFC6209]
-----+-----	-----+-----
TLS_RSA_PSK_WITH_ARIA_256_GCM_SHA384	[RFC6209]
-----+-----	-----+-----

TLS_RSA_WITH_CAMELLIA_128_GCM_SHA256	[RFC6367]
+-----+	+-----+
TLS_RSA_WITH_CAMELLIA_256_GCM_SHA384	[RFC6367]
+-----+	+-----+
TLS_RSA_PSK_WITH_CAMELLIA_128_GCM_SHA256	[RFC6367]
+-----+	+-----+
TLS_RSA_PSK_WITH_CAMELLIA_256_GCM_SHA384	[RFC6367]
+-----+	+-----+
TLS_RSA_PSK_WITH_CAMELLIA_128_CBC_SHA256	[RFC6367]
+-----+	+-----+
TLS_RSA_PSK_WITH_CAMELLIA_256_CBC_SHA384	[RFC6367]
+-----+	+-----+
TLS_RSA_WITH_AES_128_CCM	[RFC6655]
+-----+	+-----+
TLS_RSA_WITH_AES_256_CCM	[RFC6655]
+-----+	+-----+
TLS_RSA_WITH_AES_128_CCM_8	[RFC6655]
+-----+	+-----+
TLS_RSA_WITH_AES_256_CCM_8	[RFC6655]
+-----+	+-----+
TLS_RSA_PSK_WITH_CHACHA20_POLY1305_SHA256	[RFC7905]
+-----+	+-----+

Table 4

Authors' Addresses

Carrick Bartle
 Apple, Inc.
 Email: cbartle@apple.com

Nimrod Aviram
 Email: nimrod.aviram@gmail.com

TLS Working Group
Internet-Draft
Intended status: Informational
Expires: 8 September 2022

S. Celi
Cloudflare
P. Schwabe
Radboud University & MPI S&P
D. Stebila
University of Waterloo
N. Sullivan
Cloudflare
T. Wiggers
Radboud University
7 March 2022

KEM-based Authentication for TLS 1.3
draft-celi-wiggers-tls-authkem-01

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.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	3
1.1. Organization	4
2. Requirements Notation	4
3. Terminology	4
3.1. Key Encapsulation Mechanisms	4
4. Full 1.5-RTT AuthKEM Handshake Protocol	5
4.1. Client authentication	7
4.2. Relevant handshake messages	9
4.3. Overview of key differences with RFC8446 TLS 1.3	9
4.4. Implicit and explicit authentication	10
4.5. Authenticating CertificateRequest	10
5. Abbreviated AuthKEM with pre-shared public KEM keys	10
5.1. Negotiation	11
5.2. 0-RTT, forward secrecy and replay protection	12
6. Implementation	12
6.1. Negotiation of AuthKEM	13
6.2. ClientHello and ServerHello extensions	13
6.2.1. Stored Auth Key	14
6.2.2. Early authentication	15
6.3. Protocol messages	15
6.4. Cryptographic computations	16
6.4.1. Key schedule for full AuthKEM handshakes	17
6.4.2. Abbreviated AuthKEM key schedule	18
6.4.3. Computations of KEM shared secrets	19
6.4.4. Explicit Authentication Messages	19
7. Security Considerations	20
7.1. Implicit authentication	21
7.2. Authentication of Certificate Request	22
8. References	22
8.1. Normative References	22

8.2. Informative References	23
Appendix A. Acknowledgements	24
Appendix B. Open points of discussion	24
B.1. Authentication concerns for client authentication requests.	24
B.2. Interaction with signing certificates	24
Authors' Addresses	24

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:

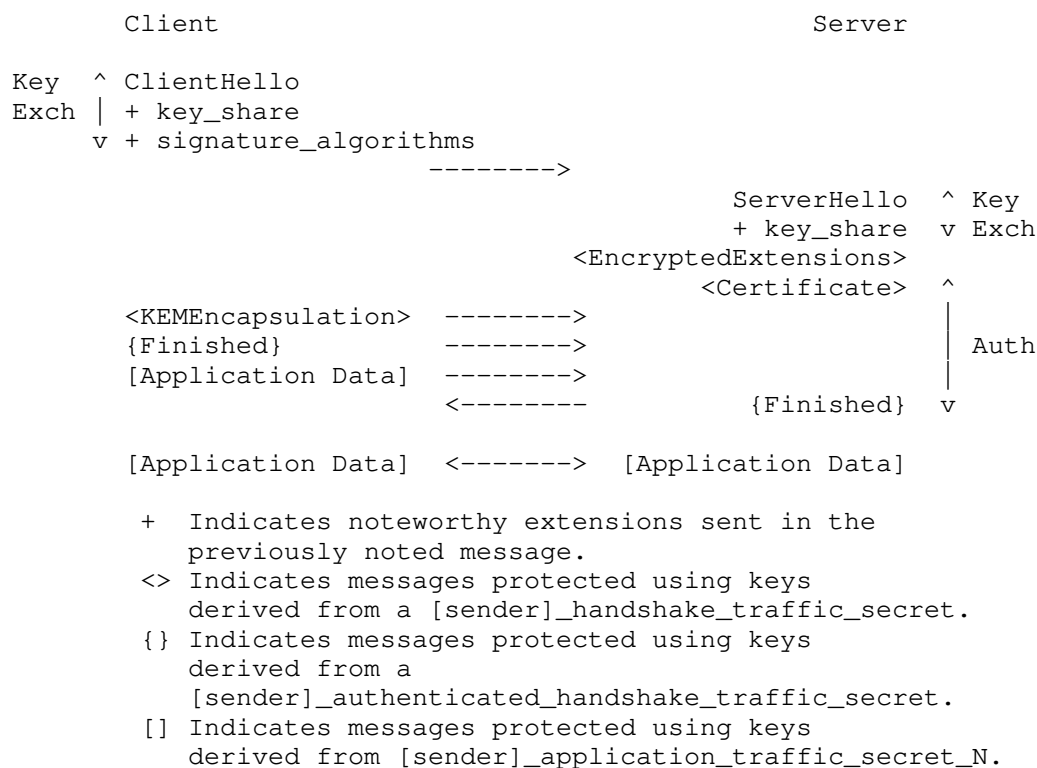


Figure 1: Message Flow for KEM-Authentication (KEM-Auth) Handshake without client authentication.

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.

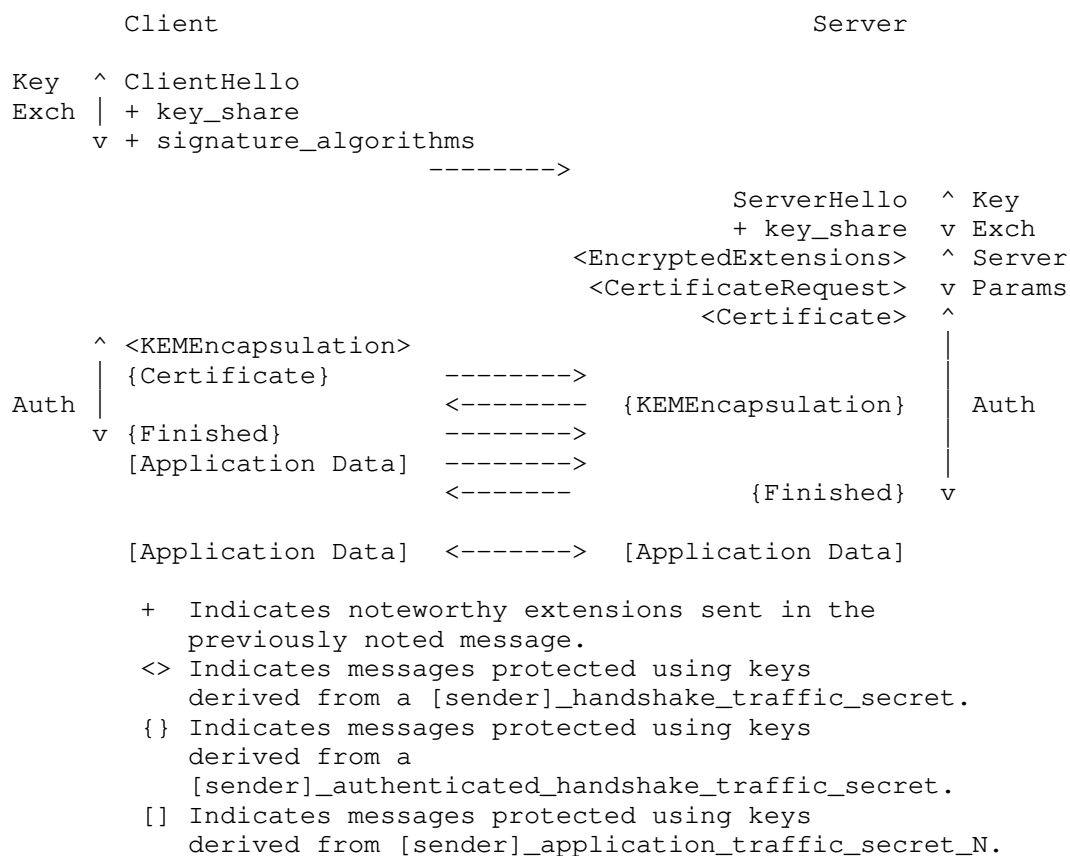


Figure 2: Message Flow for KEM-Authentication (KEM-Auth) Handshake with client authentication.

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 (<https://github.com/claucece/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?

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.

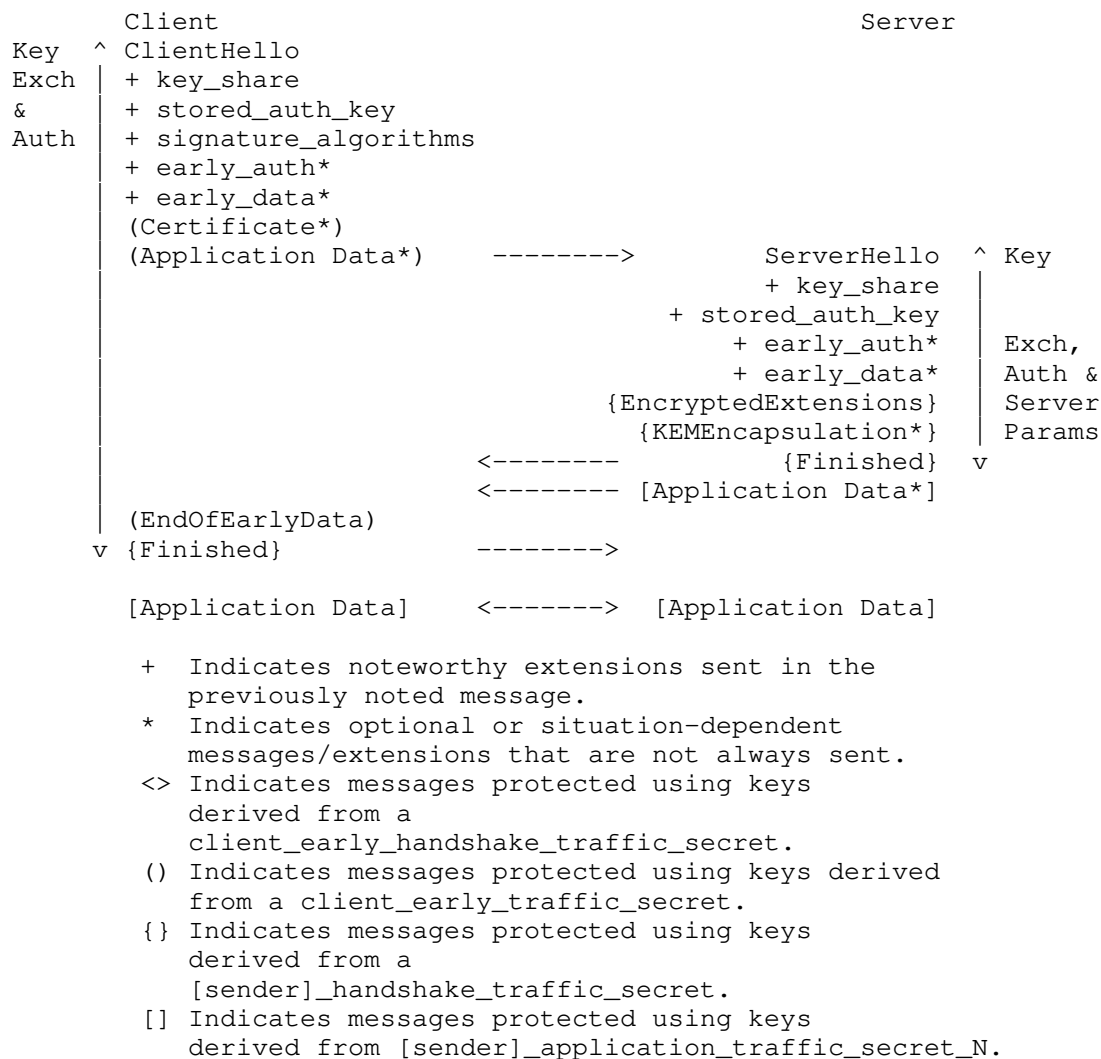


Figure 3: Abbreviated AuthKEM handshake, with optional opportunistic client authentication.

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 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 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, except 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 (<https://github.com/claucece/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.

B.2. Interaction with signing certificates

Tracked by Issue #20 (<https://github.com/claucece/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 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.

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

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 10, 2022

O. Friel
Cisco
D. Harkins
Hewlett-Packard Enterprise
July 09, 2021

Bootstrapped TLS Authentication
draft-friel-tls-eap-dpp-03

Abstract

This document defines a TLS extension that enables a server to prove to a client that it has knowledge of the public key of a key pair where the client has knowledge of the private key of the key pair. Unlike standard TLS key exchanges, the public key is never exchanged in TLS protocol messages. Proof of knowledge of the public key is used by the client to bootstrap trust in the server. The use case outlined in this document is to establish trust in an EAP server.

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 January 10, 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must

include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Bootstrap Key Pair	2
1.2. Alignment with Wi-Fi Alliance Device Provisioning Profile	3
2. Bootstrapping in TLS 1.3	4
2.1. Bootstrap Extended PSK	4
2.2. Changes to TLS 1.3 Handshake	5
3. Using TLS Bootstrapping in EAP	6
4. Summary of Work	7
5. IANA Considerations	7
6. Security Considerations	7
7. References	8
7.1. Normative References	8
7.2. Informative References	9
Authors' Addresses	9

1. Introduction

On-boarding of devices with no, or limited, user interface can be difficult. Typically, a credential is needed to access the network and network connectivity is needed to obtain a credential. This poses a catch-22.

If trust in the integrity of a device's public key can be obtained in an out-of-band fashion, a device can be authenticated and provisioned with a usable credential for network access. While this authentication can be strong, the device's authentication of the network is somewhat weaker. [duckling] presents a functional security model to address this asymmetry.

There are on-boarding protocols, such as [DPP], to address this use case but they have drawbacks. [DPP] for instance does not support wired network access. This document describes an on-boarding protocol, which we refer to as TLS Proof of Knowledge or TLS-POK.

1.1. Bootstrap Key Pair

The mechanism for on-boarding of devices defined in this document relies on bootstrap key pairs. A client device has an associated elliptic curve (EC) key pair. The key pair may be static and baked into device firmware at manufacturing time, or may be dynamic and generated at on-boarding time by the device. If this public key, specifically the ASN.1 SEQUENCE SubjectPublicKeyInfo from [RFC5280],

can be shared in a trustworthy manner with a TLS server, a form of "origin entity authentication" (the step from which all subsequent authentication proceeds) can be obtained.

The exact mechanism by which the server gains knowledge of the public key is out of scope of this specification, but possible mechanisms include scanning a QR code to obtain a base64 encoding of the ASN.1-formatted public key or upload of a Bill of Materials (BOM). If the QR code is physically attached to the client device, or the BOM is associated with the device, the assumption is that the public key obtained in this bootstrapping method belongs to the client. In this model, physical possession of the device implies legitimate ownership.

The server may have knowledge of multiple bootstrap public keys corresponding to multiple devices, and TLS extensions are defined in this document that enable the server to identify a specific bootstrap public key corresponding to a specific device.

Using the process defined herein, the client proves to the server that it has possession of the private analog to its public bootstrapping key. Provided that the mechanism in which the server obtained the bootstrapping key is trustworthy, a commensurate amount of authenticity of the resulting connection can be obtained. The server also proves that it knows the client's public key which, if the client does not gratuitously expose its public key, can be used to obtain a modicum of correctness, that the client is connecting to the correct network (see [duckling]).

1.2. Alignment with Wi-Fi Alliance Device Provisioning Profile

The definition of the bootstrap public key aligns with that given in [DPP]. This, for example, enables the QR code format as defined in [DPP] to be reused for TLS-POK. Therefore, a device that supports both wired LAN and Wi-Fi LAN connections can have a single QR code printed on its label, and the bootstrap key can be used for DPP if the device bootstraps against a Wi-Fi network, or TLS-POK if the device bootstraps against a wired network. Similarly, a common bootstrap public key format could be imported in a BOM into a server that handles devices connecting over both wired and Wi-Fi networks.

Any bootstrapping method defined for, or used by, [DPP] is compatible with TLS-POK.

2. Bootstrapping in TLS 1.3

The bootstrapping modifications introduce an extension to identify a "bootstrapping" key which is converted into an external PSK and used directly in the TLS 1.3 handshake. This key MUST be from a cryptosystem suitable for doing ECDSA.

2.1. Bootstrap Extended PSK

This document defines the "bskey" extended PSK type by expanding on the work in [extensible-psks].

```
enum {  
    bskey(TBD), (255)  
} ExtendedPskIdentityType;
```

A bskey PSK is a variant of an external PSK which, in this case, is derived from a public key.

The PSKIdentity of a bskey extended PSK is encoded with a string derived from the DER-encoded ASN.1 subjectPublicKeyInfo representation of the bootstrapping public key.

```
struct {  
    opaque identity<1..232-1>  
} BootstrapPSKIdentity;
```

Both the bskey PSK and the BootstrapPSKIdentity are computed using [RFC5869] with the hash algorithm from the ciphersuite:

```
bskeypsk = HKDF-Expand(HKDF-Extract(<>, bskey),  
                        "tls13-extended-psk-bskey", L)  
identity = HKDF-Expand(HKDF-Extract(<>, bskey),  
                        "tls13-psk-identity-bskey", L)
```

where:

- <> is a NULL salt
- bskey is the DER-encoded ASN.1 subjectPublicKeyInfo representation of the bootstrapping key
- L is the length of the digest of the underlying hash algorithm

A performance versus storage tradeoff a server can choose is to precompute the identity of every bootstrapped key with every hash algorithm that it uses in TLS and use that to quickly lookup the bootstrap key and generate the PSK. Servers that choose not to employ this optimization will have to do a runtime check with every bootstrap key it holds against the identity the client provides.

2.2. Changes to TLS 1.3 Handshake

The client includes the "tls_cert_with_extern_psk" extension in the ClientHello, per [RFC8773], and identifies the bootstrapping key using the BootstrapPSKIdentity extension. The server looks up the client's bootstrapping key in its database by checking the hash of each entry with the value received in the ClientHello. If no match is found, the server SHALL terminate the TLS handshake with an alert.

[[TODO: should we define an explicit unknown_bsk_identity alert, similar to unknown_psk_identity]]

If the server found the matching bootstrap key, it generates the bskeypsk and includes the "tls_cert_with_extern_psk" extension in the ServerHello message. When these extensions have been successfully negotiated, the TLS 1.3 key schedule SHALL include both the bskeypsk in the Early Secret derivation and an (EC)DHE shared secret value in the Handshake Secret derivation.

After successful negotiation of these extensions, the full TLS 1.3 handshake is performed with the additional caveat that the client authenticates with a raw public key (its bootstrapping key) per [RFC7250]. The bootstrapping key is always an elliptic curve public key, therefore the ClientCertTypeExtension SHALL always indicate RawPublicKey and the type of the client's Certificate SHALL be ECDSA and contain the client's bootstrapping key as a DER-encoded ASN.1 subjectPublicKeyInfo SEQUENCE.

[[DISCUSS: since the bskey identity is being negotiated we already know what the client cert type will be, the ClientCertTypeExtension is superfluous. Should it be removed from this spec?]]

When the server processes the client's Certificate it MUST ensure that it is identical to the bootstrapping public key that it used to generate an external PSK and PSKIdentifier for this handshake.

When clients use the [duckling] form of authentication, they MAY forgo the checking of the server's certificate in the CertificateVerify and rely on the integrity of the bootstrapping method employed to distribute its key in order to validate trust in the authenticated TLS connection.

The handshake is shown in Figure 1.


```

Client                                     Server
-----
ClientHello
+ bskey_id
+ cert_with_extern_psk
+ client_cert_type=RawPublicKey
+ key_share ----->
                                     ServerHello
                                     + bskey_id
                                     + cert_with_extern_psk
+ client_cert_type=RawPublicKey
+ key_share
{EncryptedExtensions}
{CertificateRequest}
{Certificate}
{CertificateVerify}
<----- {Finished}
{Certificate}
{CertificateVerify}
{Finished} ----->
[Application Data] <-----> [Application Data]

```

Figure 1: TLS 1.3 TLS-POK Handshake

3. Using TLS Bootstrapping in EAP

Enterprise deployments typically require an 802.1X/EAP-based authentication to obtain network access. Protocols like [RFC7030] can be used to enroll devices into a Certification Authority to allow them to authenticate using 802.1X/EAP. But this creates a Catch-22 where a certificate is needed for network access and network access is needed to obtain certificate.

Devices whose bootstrapping key can be obtained in an out-of-band fashion can perform an EAP-TLS-based exchange, for instance [RFC7170], and authenticate the TLS exchange using the bootstrapping extensions defined in Section 2. This network connectivity can then be used to perform an enrollment protocol (such as provided by [RFC7170]) to obtain a credential for subsequent network connectivity and certificate lifecycle maintenance.

Upon "link up", an Authenticator on an 802.1X-protected port will issue an EAP Identify request to the newly connected peer. For unprovisioned devices that desire to take advantage of TLS-POK, there is no initial realm in which to construct an NAI (see [RFC4282]) so the initial EAP Identity response SHOULD contain simply the name "TLS-POK" in order to indicate to the Authenticator that an EAP method that supports TLS-POK SHOULD be started.

Authenticating Peer	Authenticator
-----	-----
	<- EAP-Request/ Identity
EAP-Response/ Identity (TLS-POK) ->	
	<- EAP-Request/ EAP-Type=TEAP (TLS Start)
	.
	.
	.

4. Summary of Work

[TODO: agree with WG chairs where this work lives and where it should be documented.]

The protocol outlined here can be broadly broken up into 4 distinct areas:

- o TLS extensions to transport the bootstrap public key identifier
- o Use of the TLS 1.3 extension for certificate-based authentication with an external PSK
- o The client's use of a raw public key in its certificate
- o TEAP extensions to leverage the new TLS-POK handshake for trust establishment

This document captures all 4 areas, but it may be more appropriate to merge into an existing document.

5. IANA Considerations

IANA will allocated an ExtensionPSKIdentityType for the bskey type from the TLS 1.3 repository created by [extensible-psks] and replace TBD in this document with that number.

6. Security Considerations

Bootstrap and trust establishment by the TLS server is based on proof of knowledge of the client's bootstrap public key, a non-public datum. The TLS server obtains proof that the client knows its

bootstrap public key and, in addition, also possesses its corresponding private analog.

Trust on the part of the client is based on validation of the server certificate and the TLS 1.3 handshake. In addition, the client assumes that knowledge of its public bootstrapping key is not widely disseminated and therefore any device that proves knowledge of it is the appropriate device from which to receive provisioning, for instance via [RFC7170].

An attack on the bootstrapping method which substitutes the public key of a corrupted device for the public key of an honest device can result in the TLS sever on-boarding and trusting the corrupted device.

If an adversary has knowledge of the bootstrap public key, the adversary may be able to make the client bootstrap against the adversary's network. For example, if an adversary intercepts and scans QR labels on clients, and the adversary can force the client to connect to its server, then the adversary can complete the TLS-POK handshake with the client and the client will connect to the adversary's server. Since physical possession implies ownership, there is nothing to prevent a stolen device from being on-boarded.

7. References

7.1. Normative References

- [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/info/rfc5869>>.
- [RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <<https://www.rfc-editor.org/info/rfc7250>>.
- [RFC8773] Housley, R., "TLS 1.3 Extension for Certificate-Based Authentication with an External Pre-Shared Key", RFC 8773, DOI 10.17487/RFC8773, March 2020, <<https://www.rfc-editor.org/info/rfc8773>>.

7.2. Informative References

- [DPP] Wi-Fi Alliance, "Device Provisioning Profile", 2020.
- [duckling] Stajano, F. and E. Rescorla, "The Ressurecting Ducking: Security Issues for Ad-Hoc Wireless Networks", 1999.
- [extensible-psks] Wood, C. and R. Anderson, "Extensible Pre-Shared Key Types for TLS", n.d., <<https://chris-wood.github.io/draft-tls-extensible-psks/draft-group-tls-extensible-psks.html>>.
- [RFC4282] Aboba, B., Beadles, M., Arkko, J., and P. Eronen, "The Network Access Identifier", RFC 4282, DOI 10.17487/RFC4282, December 2005, <<https://www.rfc-editor.org/info/rfc4282>>.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <<https://www.rfc-editor.org/info/rfc5280>>.
- [RFC7030] Pritikin, M., Ed., Yee, P., Ed., and D. Harkins, Ed., "Enrollment over Secure Transport", RFC 7030, DOI 10.17487/RFC7030, October 2013, <<https://www.rfc-editor.org/info/rfc7030>>.
- [RFC7170] Zhou, H., Cam-Winget, N., Salowey, J., and S. Hanna, "Tunnel Extensible Authentication Protocol (TEAP) Version 1", RFC 7170, DOI 10.17487/RFC7170, May 2014, <<https://www.rfc-editor.org/info/rfc7170>>.

Authors' Addresses

Owen Friel
Cisco

Email: ofriel@cisco.com

Dan Harkins
Hewlett-Packard Enterprise

Email: daniel.harkins@hpe.com

TLS Working Group
Internet-Draft
Intended status: Standards Track
Expires: 8 September 2022

E. Rescorla
Mozilla
R. Barnes
Cisco
H. Tschofenig
Arm Limited
7 March 2022

Compact TLS 1.3
draft-ietf-tls-ctls-05

Abstract

This document specifies a "compact" version of TLS 1.3. It is isomorphic to TLS 1.3 but saves space by trimming obsolete material, tighter encoding, a template-based specialization technique, and alternative cryptographic techniques. cTLS is not directly interoperable with TLS 1.3, but it should eventually be possible for a cTLS/TLS 1.3 server to exist and successfully interoperate.

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.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components

extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	2
2. Conventions and Definitions	3
2.1. Template-based Specialization	3
2.1.1. Requirements on TLS Implementations	7
2.1.2. Predefined Extensions	7
2.1.3. Known Certificates	8
2.2. Record Layer	9
2.3. Handshake Layer	10
3. Handshake Messages	11
3.1. ClientHello	11
3.2. ServerHello	12
3.3. HelloRetryRequest	12
4. Examples	12
5. Security Considerations	13
6. IANA Considerations	13
6.1. Adding a ContentType	13
6.2. Template Keys	13
7. Normative References	14
Appendix A. Example Exchange	15
Acknowledgments	17
Authors' Addresses	17

1. Introduction

DISCLAIMER: This is a work-in-progress draft of cTLS and has not yet seen significant security analysis, so could contain major errors. It should not be used as a basis for building production systems.

This document specifies a "compact" version of TLS 1.3 [RFC8446]. It is isomorphic to TLS 1.3 but designed to take up minimal bandwidth. The space reduction is achieved by five basic techniques:

- * Omitting unnecessary values that are a holdover from previous versions of TLS.
- * Omitting the fields and handshake messages required for preserving backwards-compatibility with earlier TLS versions.
- * More compact encodings, for example point compression.

- * A template-based specialization mechanism that allows pre-populating information at both endpoints without the need for negotiation.
- * Alternative cryptographic techniques, such as semi-static Diffie-Hellman.

For the common (EC)DHE handshake with pre-established certificates, cTLS achieves an overhead of 45 bytes over the minimum required by the cryptovariables. For a PSK handshake, the overhead is 21 bytes. Annotated handshake transcripts for these cases can be found in Appendix A.

Because cTLS is semantically equivalent to TLS, it can be viewed either as a related protocol or as a compression mechanism. Specifically, it can be implemented by a layer between the TLS handshake state machine and the record layer.

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.

Structure definitions listed below override TLS 1.3 definitions; any PDU not internally defined is taken from TLS 1.3.

2.1. Template-based Specialization

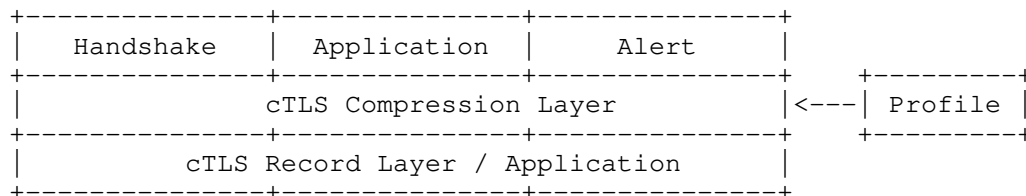
A significant transmission overhead in TLS 1.3 is contributed to by two factors, : - the negotiation of algorithm parameters, and extensions, as well as - the exchange of certificates.

TLS 1.3 supports different credential types and modes that are impacted differently by a compression scheme. For example, TLS supports certificate-based authentication, raw public key-based authentication as well as pre-shared key (PSK)-based authentication. PSK-based authentication can be used with externally configured PSKs or with PSKs established through tickets.

The basic idea of template-based specialization is that we start with the basic TLS 1.3 handshake, which is fully general and then remove degrees of freedom, eliding parts of the handshake which are used to express those degrees of freedom. For example, if we only support one version of TLS, then it is not necessary to have version negotiation and the supported_versions extension can be omitted.

Importantly, this process is performed only for the wire encoding but not for the handshake transcript. The result is that the transcript for a specialized cTLS handshake is the same as the transcript for a TLS 1.3 handshake with the same features used.

One way of thinking of this is as if specialization is a stateful compression layer between the handshake and the record layer:



By assuming that out-of-band agreements took place already prior to the start of the cTLS protocol exchange, the amount of data exchanged can be radically reduced. Because different clients may use different compression templates and because multiple compression templates may be available for use in different deployment environments, a client needs to inform the server about the profile it is planning to use. The profile field in the ClientHello serves this purpose.

Although the template-based specialization mechanisms described here are general, we also include specific mechanism for certificate-based exchanges because those are where the most complexity and size reduction can be obtained. Most of the other exchanges in TLS 1.3 are highly optimized and do not require compression to be used.

The compression profile defining the use of algorithms, algorithm parameters, and extensions is specified via a JSON dictionary.

For example, the following specialization describes a protocol with a single fixed version (TLS 1.3) and a single fixed cipher suite (TLS_AES_128_GCM_SHA256). On the wire, ClientHello.cipher_suites, ServerHello.cipher_suites, and the supported_versions extensions in the ClientHello and ServerHello would be omitted.

```
{
  "version" : 772,
  "cipherSuite" : "TLS_AES_128_GCM_SHA256"
}
```

The following elements are defined:

profile (integer): identifies the profile being defined.

`version (integer)`: indicates that both sides agree to the single TLS version specified by the given integer value (772 == 0x0304 for TLS 1.3). The `supported_versions` extension is omitted from `ClientHello.extensions` and reconstructed in the transcript as a single-valued list with the specified value. The `supported_versions` extension is omitted from `ClientHello.extensions` and reconstructed in the transcript with the specified value.

`cipherSuite (string)`: indicates that both sides agree to the single named cipher suite, using the "TLS_AEAD_HASH" syntax defined in [RFC8446], Section 8.4. The `ClientHello.cipher_suites` field is omitted and reconstructed in the transcript as a single-valued list with the specified value. The `server_hello.cipher_suite` field is omitted and reconstructed in the transcript as the specified value.

`dhGroup (string)`: specifies a single DH group to use for key establishment. The group is listed by the code point name in [RFC8446], Section 4.2.7. (e.g., `x25519`). This implies a literal "supported_groups" extension consisting solely of this group.

`signatureAlgorithm (string)`: specifies a single signature scheme to use for authentication. The signature algorithm is listed by the code point name in [RFC8446], Section 4.2.3. (e.g., `ecdsa_secp256r1_sha256`). This implies a literal "signature_algorithms" extension consisting solely of this group.

`random (integer)`: indicates that the `ClientHello.Random` and `ServerHello.Random` values are truncated to the given length. When the transcript is reconstructed, the `Random` is padded to the right with 0s and the anti-downgrade mechanism in [RFC8446], Section 4.1.3 is disabled. IMPORTANT: Using short `Random` values can lead to potential attacks. The `Random` length MUST be less than or equal to 32 bytes.

[[Open Issue: Karthik Bhargavan suggested the idea of hashing ephemeral public keys and to use the result (truncated to 32 bytes) as random values. Such a change would require a security analysis.]]

`mutualAuth (boolean)`: if set to true, indicates that the client must authenticate with a certificate by sending `Certificate` and a `CertificateVerify` message. The server MUST omit the `CertificateRequest` message, as its contents are redundant. [[OPEN ISSUE: We don't actually say that you can omit empty messages, so we need to add that somewhere.]]

`extension_order`: indicates in what order extensions appear in respective messages. This allows to omit sending the type. If there is only a single extension to be transmitted, then the extension length field can also be omitted. For example, imagine that only the KeyShare extension needs to be sent in the ClientHello as the only extension. Then, the following structure

```

28                // Extensions.length
33 26            // KeyShare
  0024          // client_shares.length
    001d        // KeyShareEntry.group
      0020 a690...af948 // KeyShareEntry.key_exchange

```

is compressed down to (assuming the KeyShare group has been pre-agreed)

```

0020 a690...af948 // KeyShareEntry.key_exchange

```

`clientHelloExtensions` (predefined extensions): Predefined ClientHello extensions, see {predefined-extensions}

`serverHelloExtensions` (predefined extensions): Predefined ServerHello extensions, see {predefined-extensions}

`encryptedExtensions` (predefined extensions): Predefined EncryptedExtensions extensions, see {predefined-extensions}

`certRequestExtensions` (predefined extensions): Predefined CertificateRequest extensions, see {predefined-extensions}

`knownCertificates` (known certificates): A compression dictionary for the Certificate message, see {known-certs}

`finishedSize` (integer): indicates that the Finished value is to be truncated to the given length. When the transcript is reconstructed, the remainder of the Finished value is filled in by the receiving side.

[[OPEN ISSUE: How short should we allow this to be? TLS 1.3 uses the native hash and TLS 1.2 used 12 bytes. More analysis is needed to know the minimum safe Finished size. See [RFC8446]; Section E.1 for more on this, as well as <https://mailarchive.ietf.org/arch/msg/tls/TugB5ddJu3nYg7chcyeIyUqWSbA.>]]

`optional` (object): contains keys that are not required to be

understood by the client. Server operators MUST NOT place a key in this section unless the server is able to determine whether the key is in use based on the client data it receives. A key MUST NOT appear in both the main template and the optional section.

2.1.1. Requirements on TLS Implementations

To be compatible with the specializations described in this section, a TLS stack needs to provide the following features:

- * If specialization of extensions is to be used, then the TLS stack MUST order each vector of Extension values in ascending order according to the ExtensionType. This allows for a deterministic reconstruction of the extension list.
- * If truncated Random values are to be used, then the TLS stack MUST be configurable to set the remaining bytes of the random values to zero. This ensures that the reconstructed, padded random value matches the original.
- * If truncated Finished values are to be used, then the TLS stack MUST be configurable so that only the provided bytes of the Finished are verified, or so that the expected remaining values can be computed.

2.1.2. Predefined Extensions

Extensions used in the ClientHello, ServerHello, EncryptedExtensions, and CertificateRequest messages can be "predefined" in a compression profile, so that they do not have to be sent on the wire. A predefined extensions object is a dictionary whose keys are extension names specified in the TLS ExtensionTypeRegistry specified in [RFC8446]. The corresponding value is a hex-encoded value for the ExtensionData field of the extension.

When compressing a handshake message, the sender compares the extensions in the message being compressed to the predefined extensions object, applying the following rules:

- * If the extensions list in the message is not sorted in ascending order by extension type, it is an error, because the decompressed message will not match.
- * If there is no entry in the predefined extensions object for the type of the extension, then the extension is included in the compressed message
- * If there is an entry:

- If the ExtensionData of the extension does not match the value in the dictionary, it is an error, because decompression will not produce the correct result.
- If the ExtensionData matches, then the extension is removed, and not included in the compressed message.

When decompressing a handshake message the receiver reconstitutes the original extensions list using the predefined extensions:

- * If there is an extension in the compressed message with a type that exists in the predefined extensions object, it is an error, because such an extension would not have been sent by a sender with a compatible compression profile.
- * For each entry in the predefined extensions dictionary, an extension is added to the decompressed message with the specified type and value.
- * The resulting vector of extensions MUST be sorted in ascending order by extension type.

Note that the "version", "dhGroup", and "signatureAlgorithm" fields in the compression profile are specific instances of this algorithm for the corresponding extensions.

[[OPEN ISSUE: Are there other extensions that would benefit from special treatment, as opposed to hex values.]]

2.1.3. Known Certificates

Certificates are a major contributor to the size of a TLS handshake. In order to avoid this overhead when the parties to a handshake have already exchanged certificates, a compression profile can specify a dictionary of "known certificates" that effectively acts as a compression dictionary on certificates.

A known certificates object is a JSON dictionary whose keys are strings containing hex-encoded compressed values. The corresponding values are hex-encoded strings representing the uncompressed values. For example:

```
{
  "00": "3082...",
  "01": "3082...",
}
```

When compressing a Certificate message, the sender examines the `cert_data` field of each `CertificateEntry`. If the `cert_data` matches a value in the known certificates object, then the sender replaces the `cert_data` with the corresponding key. Decompression works the opposite way, replacing keys with values.

Note that in this scheme, there is no signaling on the wire for whether a given `cert_data` value is compressed or uncompressed. Known certificates objects SHOULD be constructed in such a way as to avoid a uncompressed object being mistaken for compressed one and erroneously decompressed. For X.509, it is sufficient for the first byte of the compressed value (key) to have a value other than 0x30, since every X.509 certificate starts with this byte.

2.2. Record Layer

The only cTLS records that are sent in plaintext are handshake records (`ClientHello` and `ServerHello/HRR`) and alerts. cTLS alerts are the same as TLS alerts and use the same content types. For handshake records, we set the `content_type` field to a fixed cTLS-specific value to distinguish cTLS plaintext records from encrypted records, TLS/DTLS records, and other protocols using the same 5-tuple.

```
struct {  
    ContentType content_type = ctls_handshake;  
    opaque fragment<0..2^16-1>;  
} CTLSPlaintext;
```

[[OPEN ISSUE: The `profile_id` is needed in the `ClientHello` to inform the server what compression profile to use. For a `ServerHello` this field is not required. Should we make this field optional?]]

Encrypted records use DTLS [I-D.draft-ietf-tls-dtls] 1.3 record framing, comprising a configuration octet followed by optional connection ID, sequence number, and length fields. The encryption process and additional data are also as described in DTLS.

```

0 1 2 3 4 5 6 7
+---+---+---+---+---+---+
|0|0|1|C|S|L|E|E|
+---+---+---+---+---+---+
| Connection ID |      Legend:
| (if any,      |
| / length as   | C   - Connection ID (CID) present
| / negotiated) | S   - Sequence number length
+---+---+---+---+---+---+ L   - Length present
| 8 or 16 bit   | E   - Epoch
| Sequence Number
| (if present)  |
+---+---+---+---+---+---+
| 16 bit Length |
| (if present)  |
+---+---+---+---+---+---+

struct {
    opaque unified_hdr[variable];
    opaque encrypted_record[length];
} CTLSCiphertext;
```

The presence and size of the connection ID field is negotiated as in DTLS.

As with DTLS, the length field MAY be omitted by clearing the L bit, which means that the record consumes the entire rest of the data in the lower level transport. In this case it is not possible to have multiple DTLSCiphertext format records without length fields in the same datagram. In stream-oriented transports (e.g., TCP), the length field MUST be present. For use over other transports length information may be inferred from the underlying layer.

Normal DTLS does not provide a mechanism for suppressing the sequence number field entirely. When a reliable, ordered transport (e.g., TCP) is in use, the S bit in the configuration octet MUST be cleared and the sequence number MUST be omitted. When an unreliable transport is in use, the S bit has its usual meaning and the sequence number MUST be included.

2.3. Handshake Layer

The cTLS handshake framing is same as the TLS 1.3 handshake framing, except for two changes:

- * The length field is omitted.

- * The HelloRetryRequest message is a true handshake message instead of a specialization of ServerHello.

```
struct {
    HandshakeType msg_type;      /* handshake type */
    select (Handshake.msg_type) {
        case client_hello:      ClientHello;
        case server_hello:      ServerHello;
        case hello_retry_request: HelloRetryRequest;
        case end_of_early_data:  EndOfEarlyData;
        case encrypted_extensions: EncryptedExtensions;
        case certificate_request: CertificateRequest;
        case certificate:        Certificate;
        case certificate_verify: CertificateVerify;
        case finished:           Finished;
        case new_session_ticket: NewSessionTicket;
        case key_update:         KeyUpdate;
    };
} Handshake;
```

3. Handshake Messages

In general, we retain the basic structure of each individual TLS handshake message. However, the following handshake messages have been modified for space reduction and cleaned up to remove pre-TLS 1.3 baggage.

3.1. ClientHello

The cTLS ClientHello is defined as follows.

```
opaque Random[RandomLength];      // variable length

struct {
    opaque profile_id<0..2^8-1>;
    Random random;
    CipherSuite cipher_suites<1..2^16-1>;
    Extension extensions<1..2^16-1>;
} ClientHello;
```

The client uses the profile_id field to inform the server about the compression profile being used (see Section 2.1). This field MUST be set to a zero-length value and only if no compression profile is used. Non zero-length values are agreed out of band between the client and server, as part of the specification of the compression profile.

3.2. ServerHello

We redefine ServerHello in the following way.

```
struct {
    Random random;
    CipherSuite cipher_suite;
    Extension extensions<1..2^16-1>;
} ServerHello;
```

3.3. HelloRetryRequest

The HelloRetryRequest has the following format.

```
struct {
    CipherSuite cipher_suite;
    Extension extensions<2..2^16-1>;
} HelloRetryRequest;
```

The HelloRetryRequest is the same as the ServerHello above but without the unnecessary sentinel Random value.

4. Examples

This section provides some example specializations.

For this example we use TLS 1.3 only with AES_GCM, X25519, ALPN h2, short random values, and everything else is ordinary TLS 1.3.

```
{
  "profile" : 1,
  "version" : 772,
  "random": 16,
  "cipherSuite" : "TLS_AES_128_GCM_SHA256",
  "dhGroup": "X25519",
  "clientHelloExtensions": {
    "named_groups": 29,
    "application_layer_protocol_negotiation" : "030016832",
    "...": null
  }
}
```

Version 772 corresponds to the hex representation 0x0304, named group "29" (0x001D) represents X25519.

[[OPEN ISSUE: Should we have a registry of well-known profiles?]]

5. Security Considerations

WARNING: This document is effectively brand new and has seen no analysis. The idea here is that cTLS is isomorphic to TLS 1.3, and therefore should provide equivalent security guarantees.

The use of key ids is a new feature introduced in this document, which requires some analysis, especially as it looks like a potential source of identity misbinding. This is, however, entirely separable from the rest of the specification.

Transcript expansion also needs some analysis and we need to determine whether we need an extension to indicate that cTLS is in use and with which profile.

6. IANA Considerations

6.1. Adding a ContentType

This document requests that a code point be allocated from the "TLS ContentType" registry. This value must be in the range 0-31 (inclusive). The row to be added in the registry has the following form:

Value	Description	DTLS-OK	Reference
TBD	ctls	Y	RFCXXXX
TBD	ctls_handshake	Y	RFCXXXX

Table 1

[[RFC EDITOR: Please replace the value TBD with the value assigned by IANA, and the value XXXX to the RFC number assigned for this document.]]

[[OPEN ISSUE: Should we require standards action for all profile IDs that would fit in 2 octets.]]

6.2. Template Keys

This document requests that IANA open a new registry entitled "cTLS Template Keys", on the Transport Layer Security (TLS) Parameters page, with a "Specification Required" registration policy and the following initial contents:

Key	JSON Type	Reference
profile	number	(This document)
version	number	(This document)
cipherSuite	string	(This document)
dhGroup	string	(This document)
signatureAlgorithm	string	(This document)
random	number	(This document)
mutualAuth	true/false	(This document)
extension_order	object	(This document)
clientHelloExtensions	object	(This document)
serverHelloExtensions	object	(This document)
encryptedExtensions	object	(This document)
certRequestExtensions	object	(This document)
knownCertificates	object	(This document)
finishedSize	number	(This document)
optional	object	(This document)

Table 2

7. Normative References

[I-D.draft-ietf-tls-dtls]

*** BROKEN REFERENCE ***.

[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/info/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/info/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/info/rfc8446>>.

Appendix A. Example Exchange

The follow exchange illustrates a complete cTLS-based exchange supporting mutual authentication using certificates. The digital signatures use ECDSA with SHA256 and NIST P256r1. The ephemeral Diffie-Hellman uses the FX25519 curve and the exchange negotiates TLS-AES-128-CCM8-SHA256. The certificates are exchanged using certificate identifiers.

The resulting byte counts are as follows:

ECDHE			
	TLS	CTLS	Overhead
ClientHello	132	36	4
ServerHello	90	36	4
ServerFlight	478	80	7
ClientFlight	458	80	7
=====			
Total	1158	232	22

The following compression profile was used in this example:

```

{
  "profile": 1,
  "version": 772,
  "cipherSuite": "TLS_AES_128_CCM_8_SHA256",
  "dhGroup": "X25519",
  "signatureAlgorithm": "ecdsa_secp256r1_sha256",
  "finishedSize": 8,
  "clientHelloExtensions": {
    "server_name": "000e00000b6578616d706c652e636f6d",
  },
  "certificateRequestExtensions": {
    "certificate_request_context": 0,
    "signature_algorithms": "00020403"
  },
  "mutualAuth": true,
  "extension-order": {
    "clientHelloExtensions": [
      "key_share"
    ],
    "ServerHelloExtensions": [
      "key_share"
    ],
  },
  "knownCertificates": {
    "61": "3082...",
    "62": "3082...",
    "63": "...",
    "64": "...",
    ...
  }
}

```

ClientHello: 36 bytes = DH(32) + Overhead(4)

```

01          // ClientHello
01          // Profile ID
0020 a690...af948 // KeyShareEntry.key_exchange

```

ServerHello: 36 = DH(32) + Overhead(4)

```

02          // ServerHello
26          // Extensions.length
0020 9fbc...0f49 // KeyShareEntry.key_exchange

```

Server Flight: 80 = SIG(64) + MAC(8) + CERTID(1) + Overhead(7)

The EncryptedExtensions, and the CertificateRequest messages are omitted because they are empty.

```
0b          // Certificate
  03        //   CertificateList
    01      //     CertData.length
      61    //       CertData = 'a'

0f          // CertificateVerify
  4064      //   Signature.length
    3045...10ce //     Signature

14          // Finished
  bfc9d66715bb2b04 //   VerifyData
```

Client Flight: 80 bytes = SIG(64) + MAC(8) + CERTID(1) + Overhead(7)

```
0b          // Certificate
  03        //   CertificateList
    01      //     CertData.length
      62    //       CertData = 'b'

0f          // CertificateVerify
  4064      //   Signature.length
    3045...f60e //     Signature

14          // Finished
  35e9c34eec2c5dc1 //   VerifyData
```

Acknowledgments

We would like to thank Karthikeyan Bhargavan, Owen Friel, Sean Turner, Benjamin Schwartz, Martin Thomson, and Chris Wood.

Authors' Addresses

Eric Rescorla
Mozilla
Email: ekr@rtfm.com

Richard Barnes
Cisco
Email: rlb@ipv.sx

Hannes Tschofenig
Arm Limited
Email: hannes.tschofenig@arm.com

tls
Internet-Draft
Intended status: Standards Track
Expires: 17 August 2022

E. Rescorla
RTFM, Inc.
K. Oku
Fastly
N. Sullivan
C.A. Wood
Cloudflare
13 February 2022

TLS Encrypted Client Hello
draft-ietf-tls-esni-14

Abstract

This document describes a mechanism in Transport Layer Security (TLS) for encrypting a ClientHello message under a server public key.

Discussion Venues

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

Source for this draft and an issue tracker can be found at
<https://github.com/tlswg/draft-ietf-tls-esni>
(<https://github.com/tlswg/draft-ietf-tls-esni>).

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 17 August 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	3
2. Conventions and Definitions	4
3. Overview	4
3.1. Topologies	4
3.2. Encrypted ClientHello (ECH)	6
4. Encrypted ClientHello Configuration	6
4.1. Configuration Identifiers	9
4.2. Configuration Extensions	9
5. The "encrypted_client_hello" Extension	10
5.1. Encoding the ClientHelloInner	11
5.2. Authenticating the ClientHelloOuter	13
6. Client Behavior	14
6.1. Offering ECH	14
6.1.1. Encrypting the ClientHello	16
6.1.2. GREASE PSK	17
6.1.3. Recommended Padding Scheme	17
6.1.4. Determining ECH Acceptance	18
6.1.5. Handshaking with ClientHelloInner	19
6.1.6. Handshaking with ClientHelloOuter	20
6.1.7. Authenticating for the Public Name	21
6.2. GREASE ECH	22
7. Server Behavior	23
7.1. Client-Facing Server	23
7.1.1. Sending HelloRetryRequest	25
7.2. Backend Server	26
7.2.1. Sending HelloRetryRequest	27
8. Compatibility Issues	27
8.1. Misconfiguration and Deployment Concerns	28
8.2. Middleboxes	28
9. Compliance Requirements	28
10. Security Considerations	29
10.1. Security and Privacy Goals	29
10.2. Unauthenticated and Plaintext DNS	30
10.3. Client Tracking	30
10.4. Ignored Configuration Identifiers and Trial Decryption	31
10.5. Outer ClientHello	31

10.6.	Related Privacy Leaks	32
10.7.	Cookies	32
10.8.	Attacks Exploiting Acceptance Confirmation	33
10.9.	Comparison Against Criteria	33
10.9.1.	Mitigate Cut-and-Paste Attacks	34
10.9.2.	Avoid Widely Shared Secrets	34
10.9.3.	Prevent SNI-Based Denial-of-Service Attacks	34
10.9.4.	Do Not Stick Out	34
10.9.5.	Maintain Forward Secrecy	35
10.9.6.	Enable Multi-party Security Contexts	36
10.9.7.	Support Multiple Protocols	36
10.10.	Padding Policy	36
10.11.	Active Attack Mitigations	36
10.11.1.	Client Reaction Attack Mitigation	37
10.11.2.	HelloRetryRequest Hijack Mitigation	38
10.11.3.	ClientHello Malleability Mitigation	39
10.11.4.	ClientHelloInner Packet Amplification Mitigation	40
11.	IANA Considerations	41
11.1.	Update of the TLS ExtensionType Registry	41
11.2.	Update of the TLS Alert Registry	41
12.	ECHConfig Extension Guidance	41
13.	References	42
13.1.	Normative References	42
13.2.	Informative References	43
Appendix A.	Alternative SNI Protection Designs	44
A.1.	TLS-layer	44
A.1.1.	TLS in Early Data	44
A.1.2.	Combined Tickets	44
A.2.	Application-layer	45
A.2.1.	HTTP/2 CERTIFICATE Frames	45
Appendix B.	Linear-time Outer Extension Processing	45
Appendix C.	Acknowledgements	46
Appendix D.	Change Log	46
D.1.	Since draft-ietf-tls-esni-12	46
D.2.	Since draft-ietf-tls-esni-11	46
D.3.	Since draft-ietf-tls-esni-10	46
D.4.	Since draft-ietf-tls-esni-09	47
Authors' Addresses	47

1. Introduction

DISCLAIMER: This draft is work-in-progress and has not yet seen significant (or really any) security analysis. It should not be used as a basis for building production systems. This published version of the draft has been designated an "implementation draft" for testing and interop purposes.

Although TLS 1.3 [RFC8446] encrypts most of the handshake, including the server certificate, there are several ways in which an on-path attacker can learn private information about the connection. The plaintext Server Name Indication (SNI) extension in ClientHello messages, which leaks the target domain for a given connection, is perhaps the most sensitive, unencrypted information in TLS 1.3.

The target domain may also be visible through other channels, such as plaintext client DNS queries or visible server IP addresses. However, DoH [RFC8484] and DPRIVE [RFC7858] [RFC8094] provide mechanisms for clients to conceal DNS lookups from network inspection, and many TLS servers host multiple domains on the same IP address. Private origins may also be deployed behind a common provider, such as a reverse proxy. In such environments, the SNI remains the primary explicit signal used to determine the server's identity.

This document specifies a new TLS extension, called Encrypted Client Hello (ECH), that allows clients to encrypt their ClientHello to such a deployment. This protects the SNI and other potentially sensitive fields, such as the ALPN list [RFC7301]. Co-located servers with consistent externally visible TLS configurations, including supported versions and cipher suites, form an anonymity set. Usage of this mechanism reveals that a client is connecting to a particular service provider, but does not reveal which server from the anonymity set terminates the connection.

ECH is only supported with (D)TLS 1.3 [RFC8446] and newer versions of the protocol.

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. All TLS notation comes from [RFC8446], Section 3.

3. Overview

This protocol is designed to operate in one of two topologies illustrated below, which we call "Shared Mode" and "Split Mode".

3.1. Topologies

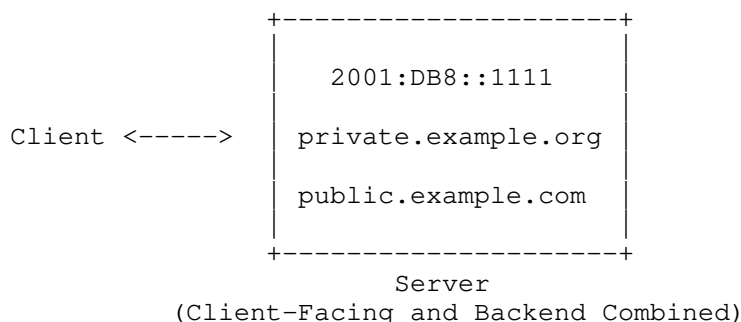


Figure 1: Shared Mode Topology

In Shared Mode, the provider is the origin server for all the domains whose DNS records point to it. In this mode, the TLS connection is terminated by the provider.

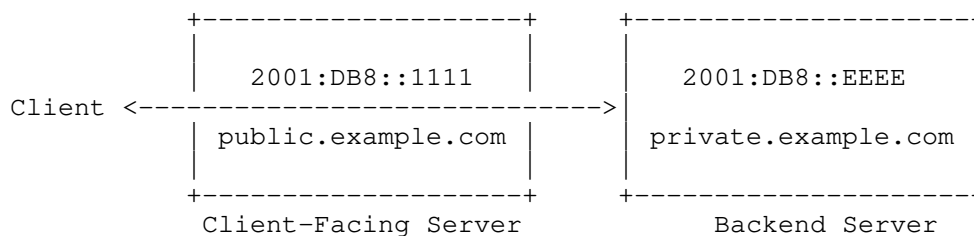


Figure 2: Split Mode Topology

In Split Mode, the provider is not the origin server for private domains. Rather, the DNS records for private domains point to the provider, and the provider's server relays the connection back to the origin server, who terminates the TLS connection with the client. Importantly, the service provider does not have access to the plaintext of the connection beyond the unencrypted portions of the handshake.

In the remainder of this document, we will refer to the ECH-service provider as the "client-facing server" and to the TLS terminator as the "backend server". These are the same entity in Shared Mode, but in Split Mode, the client-facing and backend servers are physically separated.

3.2. Encrypted ClientHello (ECH)

A client-facing server enables ECH by publishing an ECH configuration, which is an encryption public key and associated metadata. The server must publish this for all the domains it serves via Shared or Split Mode. This document defines the ECH configuration's format, but delegates DNS publication details to [HTTPS-RR]. Other delivery mechanisms are also possible. For example, the client may have the ECH configuration preconfigured.

When a client wants to establish a TLS session with some backend server, it constructs a private ClientHello, referred to as the ClientHelloInner. The client then constructs a public ClientHello, referred to as the ClientHelloOuter. The ClientHelloOuter contains innocuous values for sensitive extensions and an "encrypted_client_hello" extension (Section 5), which carries the encrypted ClientHelloInner. Finally, the client sends ClientHelloOuter to the server.

The server takes one of the following actions:

1. If it does not support ECH or cannot decrypt the extension, it completes the handshake with ClientHelloOuter. This is referred to as rejecting ECH.
2. If it successfully decrypts the extension, it forwards the ClientHelloInner to the backend server, which completes the handshake. This is referred to as accepting ECH.

Upon receiving the server's response, the client determines whether or not ECH was accepted (Section 6.1.4) and proceeds with the handshake accordingly. When ECH is rejected, the resulting connection is not usable by the client for application data. Instead, ECH rejection allows the client to retry with up-to-date configuration (Section 6.1.6).

The primary goal of ECH is to ensure that connections to servers in the same anonymity set are indistinguishable from one another. Moreover, it should achieve this goal without affecting any existing security properties of TLS 1.3. See Section 10.1 for more details about the ECH security and privacy goals.

4. Encrypted ClientHello Configuration

ECH uses HPKE for public key encryption [I-D.irtf-cfrg-hpke]. The ECH configuration is defined by the following ECHConfig structure.

```
opaque HpkePublicKey<1..2^16-1>;
uint16 HpkeKemId; // Defined in I-D.irtf-cfrg-hpke
uint16 HpkeKdfId; // Defined in I-D.irtf-cfrg-hpke
uint16 HpkeAeadId; // Defined in I-D.irtf-cfrg-hpke

struct {
    HpkeKdfId kdf_id;
    HpkeAeadId aead_id;
} HpkeSymmetricCipherSuite;

struct {
    uint8 config_id;
    HpkeKemId kem_id;
    HpkePublicKey public_key;
    HpkeSymmetricCipherSuite cipher_suites<4..2^16-4>;
} HpkeKeyConfig;

struct {
    HpkeKeyConfig key_config;
    uint8 maximum_name_length;
    opaque public_name<1..255>;
    Extension extensions<0..2^16-1>;
} ECHConfigContents;

struct {
    uint16 version;
    uint16 length;
    select (ECHConfig.version) {
        case 0xfe0d: ECHConfigContents contents;
    }
} ECHConfig;
```

The structure contains the following fields:

version The version of ECH for which this configuration is used. Beginning with draft-08, the version is the same as the code point for the "encrypted_client_hello" extension. Clients MUST ignore any ECHConfig structure with a version they do not support.

length The length, in bytes, of the next field. This length field allows implementations to skip over the elements in such a list where they cannot parse the specific version of ECHConfig.

contents An opaque byte string whose contents depend on the version. For this specification, the contents are an ECHConfigContents structure.

The ECHConfigContents structure contains the following fields:

key_config A HpkeKeyConfig structure carrying the configuration information associated with the HPKE public key. Note that this structure contains the config_id field, which applies to the entire ECHConfigContents.

maximum_name_length The longest name of a backend server, if known. If not known, this value can be set to zero. It is used to compute padding (Section 6.1.3) and does not constrain server name lengths. Names may exceed this length if, e.g., the server uses wildcard names or added new names to the anonymity set.

public_name The DNS name of the client-facing server, i.e., the entity trusted to update the ECH configuration. This is used to correct misconfigured clients, as described in Section 6.1.6.

Clients MUST ignore any ECHConfig structure whose public_name is not parsable as a dot-separated sequence of LDH labels, as defined in [RFC5890], Section 2.3.1 or which begins or end with an ASCII dot.

Clients SHOULD ignore the ECHConfig if it contains an encoded IPv4 address. To determine if a public_name value is an IPv4 address, clients can invoke the IPv4 parser algorithm in [WHATWG-IPV4]. It returns a value when the input is an IPv4 address.

See Section 6.1.7 for how the client interprets and validates the public_name.

extensions A list of extensions that the client must take into consideration when generating a ClientHello message. These are described below (Section 4.2).

[[OPEN ISSUE: determine if clients should enforce a 63-octet label limit for public_name]] [[OPEN ISSUE: fix reference to WHATWG-IPV4]]

The HpkeKeyConfig structure contains the following fields:

config_id A one-byte identifier for the given HPKE key configuration. This is used by clients to indicate the key used for ClientHello encryption. Section 4.1 describes how client-facing servers allocate this value.

kem_id The HPKE KEM identifier corresponding to public_key. Clients MUST ignore any ECHConfig structure with a key using a KEM they do not support.

public_key The HPKE public key used by the client to encrypt ClientHelloInner.

`cipher_suites` The list of HPKE KDF and AEAD identifier pairs clients can use for encrypting `ClientHelloInner`. See Section 6.1 for how clients choose from this list.

The client-facing server advertises a sequence of ECH configurations to clients, serialized as follows.

```
ECHConfig ECHConfigList<1..2^16-1>;
```

The `ECHConfigList` structure contains one or more `ECHConfig` structures in decreasing order of preference. This allows a server to support multiple versions of ECH and multiple sets of ECH parameters.

4.1. Configuration Identifiers

A client-facing server has a set of known `ECHConfig` values, with corresponding private keys. This set SHOULD contain the currently published values, as well as previous values that may still be in use, since clients may cache DNS records up to a TTL or longer.

Section 7.1 describes a trial decryption process for decrypting the `ClientHello`. This can impact performance when the client-facing server maintains many known `ECHConfig` values. To avoid this, the client-facing server SHOULD allocate distinct `config_id` values for each `ECHConfig` in its known set. The RECOMMENDED strategy is via rejection sampling, i.e., to randomly select `config_id` repeatedly until it does not match any known `ECHConfig`.

It is not necessary for `config_id` values across different client-facing servers to be distinct. A backend server may be hosted behind two different client-facing servers with colliding `config_id` values without any performance impact. Values may also be reused if the previous `ECHConfig` is no longer in the known set.

4.2. Configuration Extensions

ECH configuration extensions are used to provide room for additional functionality as needed. See Section 12 for guidance on which types of extensions are appropriate for this structure.

The format is as defined in [RFC8446], Section 4.2. The same interpretation rules apply: extensions MAY appear in any order, but there MUST NOT be more than one extension of the same type in the extensions block. An extension can be tagged as mandatory by using an extension type codepoint with the high order bit set to 1.

Clients MUST parse the extension list and check for unsupported mandatory extensions. If an unsupported mandatory extension is present, clients MUST ignore the ECHConfig.

5. The "encrypted_client_hello" Extension

To offer ECH, the client sends an "encrypted_client_hello" extension in the ClientHelloOuter. When it does, it MUST also send the extension in ClientHelloInner.

```
enum {  
    encrypted_client_hello(0xfe0d), (65535)  
} ExtensionType;
```

The payload of the extension has the following structure:

```
enum { outer(0), inner(1) } ECHClientHelloType;  
  
struct {  
    ECHClientHelloType type;  
    select (ECHClientHello.type) {  
        case outer:  
            HpkeSymmetricCipherSuite cipher_suite;  
            uint8 config_id;  
            opaque enc<0..2^16-1>;  
            opaque payload<1..2^16-1>;  
        case inner:  
            Empty;  
    };  
} ECHClientHello;
```

The outer extension uses the outer variant and the inner extension uses the inner variant. The inner extension has an empty payload. The outer extension has the following fields:

config_id The ECHConfigContents.key_config.config_id for the chosen ECHConfig.

cipher_suite The cipher suite used to encrypt ClientHelloInner. This MUST match a value provided in the corresponding ECHConfigContents.cipher_suites list.

enc The HPKE encapsulated key, used by servers to decrypt the corresponding payload field. This field is empty in a ClientHelloOuter sent in response to HelloRetryRequest.

payload The serialized and encrypted ClientHelloInner structure, encrypted using HPKE as described in Section 6.1.

When a client offers the outer version of an "encrypted_client_hello" extension, the server MAY include an "encrypted_client_hello" extension in its EncryptedExtensions message, as described in Section 7.1, with the following payload:

```
struct {  
    ECHConfigList retry_configs;  
} ECHEncryptedExtensions;
```

The response is valid only when the server used the ClientHelloOuter. If the server sent this extension in response to the inner variant, then the client MUST abort with an "unsupported_extension" alert.

`retry_configs` An ECHConfigList structure containing one or more ECHConfig structures, in decreasing order of preference, to be used by the client as described in Section 6.1.6. These are known as the server's "retry configurations".

Finally, when the client offers the "encrypted_client_hello", if the payload is the inner variant and the server responds with HelloRetryRequest, it MUST include an "encrypted_client_hello" extension with the following payload:

```
struct {  
    opaque confirmation[8];  
} ECHHelloRetryRequest;
```

The value of ECHHelloRetryRequest.confirmation is set to `hrr_accept_confirmation` as described in Section 7.2.1.

This document also defines the "ech_required" alert, which the client MUST send when it offered an "encrypted_client_hello" extension that was not accepted by the server. (See Section 11.2.)

5.1. Encoding the ClientHelloInner

Before encrypting, the client pads and optionally compresses ClientHelloInner into a EncodedClientHelloInner structure, defined below:

```
struct {  
    ClientHello client_hello;  
    uint8 zeros[length_of_padding];  
} EncodedClientHelloInner;
```

The `client_hello` field is computed by first making a copy of `ClientHelloInner` and setting the `legacy_session_id` field to the empty string. Note this field uses the `ClientHello` structure, defined in Section 4.1.2 of [RFC8446] which does not include the Handshake structure's four byte header. The `zeros` field MUST be all zeroes.

Repeating large extensions, such as "key_share" with post-quantum algorithms, between `ClientHelloInner` and `ClientHelloOuter` can lead to excessive size. To reduce the size impact, the client MAY substitute extensions which it knows will be duplicated in `ClientHelloOuter`. It does so by removing and replacing extensions from `EncodedClientHelloInner` with a single "ech_outer_extensions" extension, defined as follows:

```
enum {  
    ech_outer_extensions(0xfd00), (65535)  
} ExtensionType;
```

```
ExtensionType OuterExtensions<2..254>;
```

`OuterExtensions` contains the removed `ExtensionType` values. Each value references the matching extension in `ClientHelloOuter`. The values MUST be ordered contiguously in `ClientHelloInner`, and the "ech_outer_extensions" extension MUST be inserted in the corresponding position in `EncodedClientHelloInner`. Additionally, the extensions MUST appear in `ClientHelloOuter` in the same relative order. However, there is no requirement that they be contiguous. For example, `OuterExtensions` may contain extensions A, B, C, while `ClientHelloOuter` contains extensions A, D, B, C, E, F.

The "ech_outer_extensions" extension can only be included in `EncodedClientHelloInner`, and MUST NOT appear in either `ClientHelloOuter` or `ClientHelloInner`.

Finally, the client pads the message by setting the `zeros` field to a byte string whose contents are all zeros and whose length is the amount of padding to add. Section 6.1.3 describes a recommended padding scheme.

The client-facing server computes `ClientHelloInner` by reversing this process. First it parses `EncodedClientHelloInner`, interpreting all bytes after `client_hello` as padding. If any padding byte is non-zero, the server MUST abort the connection with an "illegal_parameter" alert.

Next it makes a copy of the `client_hello` field and copies the `legacy_session_id` field from `ClientHelloOuter`. It then looks for an "ech_outer_extensions" extension. If found, it replaces the

extension with the corresponding sequence of extensions in the ClientHelloOuter. The server MUST abort the connection with an "illegal_parameter" alert if any of the following are true:

- * Any referenced extension is missing in ClientHelloOuter.
- * Any extension is referenced in OuterExtensions more than once.
- * "encrypted_client_hello" is referenced in OuterExtensions.
- * The extensions in ClientHelloOuter corresponding to those in OuterExtensions do not occur in the same order.

These requirements prevent an attacker from performing a packet amplification attack, by crafting a ClientHelloOuter which decompresses to a much larger ClientHelloInner. This is discussed further in Section 10.11.4.

Implementations SHOULD bound the time to compute a ClientHelloInner proportionally to the ClientHelloOuter size. If the cost is disproportionately large, a malicious client could exploit this in a denial of service attack. Appendix B describes a linear-time procedure that may be used for this purpose.

5.2. Authenticating the ClientHelloOuter

To prevent a network attacker from modifying the reconstructed ClientHelloInner (see Section 10.11.3), ECH authenticates ClientHelloOuter by passing ClientHelloOuterAAD as the associated data for HPKE sealing and opening operations. The ClientHelloOuterAAD is a serialized ClientHello structure, defined in Section 4.1.2 of [RFC8446], which matches the ClientHelloOuter except the payload field of the "encrypted_client_hello" is replaced with a byte string of the same length but whose contents are zeros. This value does not include the four-byte header from the Handshake structure.

The client follows the procedure in Section 6.1.1 to first construct ClientHelloOuterAAD with a placeholder payload field, then replace the field with the encrypted value to compute ClientHelloOuter.

The server then receives ClientHelloOuter and computes ClientHelloOuterAAD by making a copy and replacing the portion corresponding to the payload field with zeros.

The payload and the placeholder strings have the same length, so it is not necessary for either side to recompute length prefixes when applying the above transformations.

The decompression process in Section 5.1 forbids "encrypted_client_hello" in OuterExtensions. This ensures the unauthenticated portion of ClientHelloOuter is not incorporated into ClientHelloInner.

6. Client Behavior

Clients that implement the ECH extension behave in one of two ways: either they offer a real ECH extension, as described in Section 6.1; or they send a GREASE ECH extension, as described in Section 6.2. Clients of the latter type do not negotiate ECH. Instead, they generate a dummy ECH extension that is ignored by the server. (See Section 10.9.4 for an explanation.) The client offers ECH if it is in possession of a compatible ECH configuration and sends GREASE ECH otherwise.

6.1. Offering ECH

To offer ECH, the client first chooses a suitable ECHConfig from the server's ECHConfigList. To determine if a given ECHConfig is suitable, it checks that it supports the KEM algorithm identified by ECHConfig.contents.kem_id, at least one KDF/AEAD algorithm identified by ECHConfig.contents.cipher_suites, and the version of ECH indicated by ECHConfig.contents.version. Once a suitable configuration is found, the client selects the cipher suite it will use for encryption. It MUST NOT choose a cipher suite or version not advertised by the configuration. If no compatible configuration is found, then the client SHOULD proceed as described in Section 6.2.

Next, the client constructs the ClientHelloInner message just as it does a standard ClientHello, with the exception of the following rules:

1. It MUST NOT offer to negotiate TLS 1.2 or below. This is necessary to ensure the backend server does not negotiate a TLS version that is incompatible with ECH.
2. It MUST NOT offer to resume any session for TLS 1.2 and below.
3. If it intends to compress any extensions (see Section 5.1), it MUST order those extensions consecutively.
4. It MUST include the "encrypted_client_hello" extension of type inner as described in Section 5. (This requirement is not applicable when the "encrypted_client_hello" extension is generated as described in Section 6.2.)

The client then constructs `EncodedClientHelloInner` as described in Section 5.1. It also computes an HPKE encryption context and enc value as:

```
pkR = DeserializePublicKey(ECHConfig.contents.public_key)
enc, context = SetupBaseS(pkR,
                          "tls ech" || 0x00 || ECHConfig)
```

Next, it constructs a partial `ClientHelloOuterAAD` as it does a standard `ClientHello`, with the exception of the following rules:

1. It MUST offer to negotiate TLS 1.3 or above.
2. If it compressed any extensions in `EncodedClientHelloInner`, it MUST copy the corresponding extensions from `ClientHelloInner`. The copied extensions additionally MUST be in the same relative order as in `ClientHelloInner`.
3. It MUST copy the `legacy_session_id` field from `ClientHelloInner`. This allows the server to echo the correct session ID for TLS 1.3's compatibility mode (see Appendix D.4 of [RFC8446]) when ECH is negotiated.
4. It MAY copy any other field from the `ClientHelloInner` except `ClientHelloInner.random`. Instead, It MUST generate a fresh `ClientHelloOuter.random` using a secure random number generator. (See Section 10.11.1.)
5. The value of `ECHConfig.contents.public_name` MUST be placed in the "server_name" extension.
6. When the client offers the "pre_shared_key" extension in `ClientHelloInner`, it SHOULD also include a GREASE "pre_shared_key" extension in `ClientHelloOuter`, generated in the manner described in Section 6.1.2. The client MUST NOT use this extension to advertise a PSK to the client-facing server. (See Section 10.11.3.) When the client includes a GREASE "pre_shared_key" extension, it MUST also copy the "psk_key_exchange_modes" from the `ClientHelloInner` into the `ClientHelloOuter`.
7. When the client offers the "early_data" extension in `ClientHelloInner`, it MUST also include the "early_data" extension in `ClientHelloOuter`. This allows servers that reject ECH and use `ClientHelloOuter` to safely ignore any early data sent by the client per [RFC8446], Section 4.2.10.

Note that these rules may change in the presence of an application profile specifying otherwise.

The client might duplicate non-sensitive extensions in both messages. However, implementations need to take care to ensure that sensitive extensions are not offered in the ClientHelloOuter. See Section 10.5 for additional guidance.

Finally, the client encrypts the EncodedClientHelloInner with the above values, as described in Section 6.1.1, to construct a ClientHelloOuter. It sends this to the server, and processes the response as described in Section 6.1.4.

6.1.1. Encrypting the ClientHello

Given an EncodedClientHelloInner, an HPKE encryption context and enc value, and a partial ClientHelloOuterAAD, the client constructs a ClientHelloOuter as follows.

First, the client determines the length *L* of encrypting EncodedClientHelloInner with the selected HPKE AEAD. This is typically the sum of the plaintext length and the AEAD tag length. The client then completes the ClientHelloOuterAAD with an "encrypted_client_hello" extension. This extension value contains the outer variant of ECHClientHello with the following fields:

- * *config_id*, the identifier corresponding to the chosen ECHConfig structure;
- * *cipher_suite*, the client's chosen cipher suite;
- * *enc*, as given above; and
- * *payload*, a placeholder byte string containing *L* zeros.

If configuration identifiers (see Section 10.4) are to be ignored, *config_id* SHOULD be set to a randomly generated byte in the first ClientHelloOuter and, in the event of HRR, MUST be left unchanged for the second ClientHelloOuter.

The client serializes this structure to construct the ClientHelloOuterAAD. It then computes the final payload as:

```
final_payload = context.Seal(ClientHelloOuterAAD,  
                             EncodedClientHelloInner)
```

Finally, the client replaces payload with final_payload to obtain ClientHelloOuter. The two values have the same length, so it is not necessary to recompute length prefixes in the serialized structure.

Note this construction requires the "encrypted_client_hello" be computed after all other extensions. This is possible because the ClientHelloOuter's "pre_shared_key" extension is either omitted, or uses a random binder (Section 6.1.2).

6.1.2. GREASE PSK

When offering ECH, the client is not permitted to advertise PSK identities in the ClientHelloOuter. However, the client can send a "pre_shared_key" extension in the ClientHelloInner. In this case, when resuming a session with the client, the backend server sends a "pre_shared_key" extension in its ServerHello. This would appear to a network observer as if the the server were sending this extension without solicitation, which would violate the extension rules described in [RFC8446]. Sending a GREASE "pre_shared_key" extension in the ClientHelloOuter makes it appear to the network as if the extension were negotiated properly.

The client generates the extension payload by constructing an OfferedPsks structure (see [RFC8446], Section 4.2.11) as follows. For each PSK identity advertised in the ClientHelloInner, the client generates a random PSK identity with the same length. It also generates a random, 32-bit, unsigned integer to use as the obfuscated_ticket_age. Likewise, for each inner PSK binder, the client generates a random string of the same length.

Per the rules of Section 6.1, the server is not permitted to resume a connection in the outer handshake. If ECH is rejected and the client-facing server replies with a "pre_shared_key" extension in its ServerHello, then the client MUST abort the handshake with an "illegal_parameter" alert.

6.1.3. Recommended Padding Scheme

This section describes a deterministic padding mechanism based on the following observation: individual extensions can reveal sensitive information through their length. Thus, each extension in the inner ClientHello may require different amounts of padding. This padding may be fully determined by the client's configuration or may require server input.

By way of example, clients typically support a small number of application profiles. For instance, a browser might support HTTP with ALPN values ["http/1.1", "h2"] and WebRTC media with ALPNs

["webrtc", "c-webrtc"]. Clients SHOULD pad this extension by rounding up to the total size of the longest ALPN extension across all application profiles. The target padding length of most ClientHello extensions can be computed in this way.

In contrast, clients do not know the longest SNI value in the client-facing server's anonymity set without server input. Clients SHOULD use the ECHConfig's maximum_name_length field as follows, where L is the maximum_name_length value.

1. If the ClientHelloInner contained a "server_name" extension with a name of length D, add $\max(0, L - D)$ bytes of padding.
2. If the ClientHelloInner did not contain a "server_name" extension (e.g., if the client is connecting to an IP address), add $L + 9$ bytes of padding. This is the length of a "server_name" extension with an L-byte name.

Finally, the client SHOULD pad the entire message as follows:

1. Let L be the length of the EncodedClientHelloInner with all the padding computed so far.
2. Let $N = 31 - ((L - 1) \% 32)$ and add N bytes of padding.

This rounds the length of EncodedClientHelloInner up to a multiple of 32 bytes, reducing the set of possible lengths across all clients.

In addition to padding ClientHelloInner, clients and servers will also need to pad all other handshake messages that have sensitive-length fields. For example, if a client proposes ALPN values in ClientHelloInner, the server-selected value will be returned in an EncryptedExtension, so that handshake message also needs to be padded using TLS record layer padding.

6.1.4. Determining ECH Acceptance

As described in Section 7, the server may either accept ECH and use ClientHelloInner or reject it and use ClientHelloOuter. This is determined by the server's initial message.

If the message does not negotiate TLS 1.3 or higher, the server has rejected ECH. Otherwise, it is either a ServerHello or HelloRetryRequest.

If the message is a `ServerHello`, the client computes `accept_confirmation` as described in Section 7.2. If this value matches the last 8 bytes of `ServerHello.random`, the server has accepted ECH. Otherwise, it has rejected ECH.

If the message is a `HelloRetryRequest`, the client checks for the `"encrypted_client_hello"` extension. If none is found, the server has rejected ECH. Otherwise, if it has a length other than 8, the client aborts the handshake with a `"decode_error"` alert. Otherwise, the client computes `hrr_accept_confirmation` as described in Section 7.2.1. If this value matches the extension payload, the server has accepted ECH. Otherwise, it has rejected ECH.

[[OPEN ISSUE: Depending on what we do for issue#450, it may be appropriate to change the client behavior if the HRR extension is present but with the wrong value.]]

If the server accepts ECH, the client handshakes with `ClientHelloInner` as described in Section 6.1.5. Otherwise, the client handshakes with `ClientHelloOuter` as described in Section 6.1.6.

6.1.5. Handshaking with `ClientHelloInner`

If the server accepts ECH, the client proceeds with the connection as in [RFC8446], with the following modifications:

The client behaves as if it had sent `ClientHelloInner` as the `ClientHello`. That is, it evaluates the handshake using the `ClientHelloInner`'s preferences, and, when computing the transcript hash (Section 4.4.1 of [RFC8446]), it uses `ClientHelloInner` as the first `ClientHello`.

If the server responds with a `HelloRetryRequest`, the client computes the updated `ClientHello` message as follows:

1. It computes a second `ClientHelloInner` based on the first `ClientHelloInner`, as in Section 4.1.4 of [RFC8446]. The `ClientHelloInner`'s `"encrypted_client_hello"` extension is left unmodified.
2. It constructs `EncodedClientHelloInner` as described in Section 5.1.

3. It constructs a second partial ClientHelloOuterAAD message. This message MUST be syntactically valid. The extensions MAY be copied from the original ClientHelloOuter unmodified, or omitted. If not sensitive, the client MAY copy updated extensions from the second ClientHelloInner for compression.
4. It encrypts EncodedClientHelloInner as described in Section 6.1.1, using the second partial ClientHelloOuterAAD, to obtain a second ClientHelloOuter. It reuses the original HPKE encryption context computed in Section 6.1 and uses the empty string for enc.

The HPKE context maintains a sequence number, so this operation internally uses a fresh nonce for each AEAD operation. Reusing the HPKE context avoids an attack described in Section 10.11.2.

The client then sends the second ClientHelloOuter to the server. However, as above, it uses the second ClientHelloInner for preferences, and both the ClientHelloInner messages for the transcript hash. Additionally, it checks the resulting ServerHello for ECH acceptance as in Section 6.1.4. If the ServerHello does not also indicate ECH acceptance, the client MUST terminate the connection with an "illegal_parameter" alert.

6.1.6. Handshaking with ClientHelloOuter

If the server rejects ECH, the client proceeds with the handshake, authenticating for ECHConfig.contents.public_name as described in Section 6.1.7. If authentication or the handshake fails, the client MUST return a failure to the calling application. It MUST NOT use the retry configurations. It MUST NOT treat this as a secure signal to disable ECH.

If the server supplied an "encrypted_client_hello" extension in its EncryptedExtensions message, the client MUST check that it is syntactically valid and the client MUST abort the connection with a "decode_error" alert otherwise. If an earlier TLS version was negotiated, the client MUST NOT enable the False Start optimization [RFC7918] for this handshake. If both authentication and the handshake complete successfully, the client MUST perform the processing described below then abort the connection with an "ech_required" alert before sending any application data to the server.

If the server provided "retry_configs" and if at least one of the values contains a version supported by the client, the client can regard the ECH keys as securely replaced by the server. It SHOULD retry the handshake with a new transport connection, using the retry

configurations supplied by the server. The retry configurations may only be applied to the retry connection. The client MUST NOT use retry configurations for connections beyond the retry. This avoids introducing pinning concerns or a tracking vector, should a malicious server present client-specific retry configurations in order to identify the client in a subsequent ECH handshake.

If none of the values provided in "retry_configs" contains a supported version, or an earlier TLS version was negotiated, the client can regard ECH as securely disabled by the server, and it SHOULD retry the handshake with a new transport connection and ECH disabled.

Clients SHOULD implement a limit on retries caused by receipt of "retry_configs" or servers which do not acknowledge the "encrypted_client_hello" extension. If the client does not retry in either scenario, it MUST report an error to the calling application.

6.1.7. Authenticating for the Public Name

When the server rejects ECH, it continues with the handshake using the plaintext "server_name" extension instead (see Section 7). Clients that offer ECH then authenticate the connection with the public name, as follows:

- * The client MUST verify that the certificate is valid for ECHConfig.contents.public_name. If invalid, it MUST abort the connection with the appropriate alert.
- * If the server requests a client certificate, the client MUST respond with an empty Certificate message, denoting no client certificate.

In verifying the client-facing server certificate, the client MUST interpret the public name as a DNS-based reference identity. Clients that incorporate DNS names and IP addresses into the same syntax (e.g. [RFC3986], Section 7.4 and [WHATWG-IPV4]) MUST reject names that would be interpreted as IPv4 addresses. Clients that enforce this by checking and rejecting encoded IPv4 addresses in ECHConfig.contents.public_name do not need to repeat the check at this layer.

Note that authenticating a connection for the public name does not authenticate it for the origin. The TLS implementation MUST NOT report such connections as successful to the application. It additionally MUST ignore all session tickets and session IDs presented by the server. These connections are only used to trigger retries, as described in Section 6.1.6. This may be implemented, for instance, by reporting a failed connection with a dedicated error code.

6.2. GREASE ECH

If the client attempts to connect to a server and does not have an ECHConfig structure available for the server, it SHOULD send a GREASE [RFC8701] "encrypted_client_hello" extension in the first ClientHello as follows:

- * Set the config_id field to a random byte.
- * Set the cipher_suite field to a supported HpkeSymmetricCipherSuite. The selection SHOULD vary to exercise all supported configurations, but MAY be held constant for successive connections to the same server in the same session.
- * Set the enc field to a randomly-generated valid encapsulated public key output by the HPKE KEM.
- * Set the payload field to a randomly-generated string of L+C bytes, where C is the ciphertext expansion of the selected AEAD scheme and L is the size of the EncodedClientHelloInner the client would compute when offering ECH, padded according to Section 6.1.3.

If sending a second ClientHello in response to a HelloRetryRequest, the client copies the entire "encrypted_client_hello" extension from the first ClientHello. The identical value will reveal to an observer that the value of "encrypted_client_hello" was fake, but this only occurs if there is a HelloRetryRequest.

If the server sends an "encrypted_client_hello" extension in either HelloRetryRequest or EncryptedExtensions, the client MUST check the extension syntactically and abort the connection with a "decode_error" alert if it is invalid. It otherwise ignores the extension. It MUST NOT save the "retry_config" value in EncryptedExtensions.

Offering a GREASE extension is not considered offering an encrypted ClientHello for purposes of requirements in Section 6.1. In particular, the client MAY offer to resume sessions established without ECH.

7. Server Behavior

Servers that support ECH play one of two roles, depending on the payload of the "encrypted_client_hello" extension in the initial ClientHello:

- * If ECHClientHello.type is outer, then the server acts as a client-facing server and proceeds as described in Section 7.1 to extract a ClientHelloInner, if available.
- * If ECHClientHello.type is inner, then the server acts as a backend server and proceeds as described in Section 7.2.
- * Otherwise, if ECHClientHello.type is not a valid ECHClientHelloType, then the server MUST abort with an "illegal_parameter" alert.

If the "encrypted_client_hello" is not present, then the server completes the handshake normally, as described in [RFC8446].

7.1. Client-Facing Server

Upon receiving an "encrypted_client_hello" extension in an initial ClientHello, the client-facing server determines if it will accept ECH, prior to negotiating any other TLS parameters. Note that successfully decrypting the extension will result in a new ClientHello to process, so even the client's TLS version preferences may have changed.

First, the server collects a set of candidate ECHConfig values. This list is determined by one of the two following methods:

1. Compare ECHClientHello.config_id against identifiers of each known ECHConfig and select the ones that match, if any, as candidates.
2. Collect all known ECHConfig values as candidates, with trial decryption below determining the final selection.

Some uses of ECH, such as local discovery mode, may randomize the ECHClientHello.config_id since it can be used as a tracking vector. In such cases, the second method should be used for matching the ECHClientHello to a known ECHConfig. See Section 10.4. Unless specified by the application profile or otherwise externally configured, implementations MUST use the first method.

The server then iterates over the candidate ECHConfig values, attempting to decrypt the "encrypted_client_hello" extension:

The server verifies that the ECHConfig supports the cipher suite indicated by the ECHClientHello.cipher_suite and that the version of ECH indicated by the client matches the ECHConfig.version. If not, the server continues to the next candidate ECHConfig.

Next, the server decrypts ECHClientHello.payload, using the private key skR corresponding to ECHConfig, as follows:

```
context = SetupBaseR(ECHClientHello.enc, skR,  
                    "tls ech" || 0x00 || ECHConfig)  
EncodedClientHelloInner = context.Open(ClientHelloOuterAAD,  
                                       ECHClientHello.payload)
```

ClientHelloOuterAAD is computed from ClientHelloOuter as described in Section 5.2. The info parameter to SetupBaseR is the concatenation "tls ech", a zero byte, and the serialized ECHConfig. If decryption fails, the server continues to the next candidate ECHConfig. Otherwise, the server reconstructs ClientHelloInner from EncodedClientHelloInner, as described in Section 5.1. It then stops iterating over the candidate ECHConfig values.

Upon determining the ClientHelloInner, the client-facing server checks that the message includes a well-formed "encrypted_client_hello" extension of type inner and that it does not offer TLS 1.2 or below. If either of these checks fails, the client-facing server MUST abort with an "illegal_parameter" alert.

If these checks succeed, the client-facing server then forwards the ClientHelloInner to the appropriate backend server, which proceeds as in Section 7.2. If the backend server responds with a HelloRetryRequest, the client-facing server forwards it, decrypts the client's second ClientHelloOuter using the procedure in Section 7.1.1, and forwards the resulting second ClientHelloInner. The client-facing server forwards all other TLS messages between the client and backend server unmodified.

Otherwise, if all candidate ECHConfig values fail to decrypt the extension, the client-facing server MUST ignore the extension and proceed with the connection using ClientHelloOuter, with the following modifications:

- * If sending a HelloRetryRequest, the server MAY include an "encrypted_client_hello" extension with a payload of 8 random bytes; see Section 10.9.4 for details.
- * If the server is configured with any ECHConfigs, it MUST include the "encrypted_client_hello" extension in its EncryptedExtensions with the "retry_configs" field set to one or more ECHConfig

structures with up-to-date keys. Servers MAY supply multiple ECHConfig values of different versions. This allows a server to support multiple versions at once.

Note that decryption failure could indicate a GREASE ECH extension (see Section 6.2), so it is necessary for servers to proceed with the connection and rely on the client to abort if ECH was required. In particular, the unrecognized value alone does not indicate a misconfigured ECH advertisement (Section 8.1). Instead, servers can measure occurrences of the "ech_required" alert to detect this case.

7.1.1. Sending HelloRetryRequest

After sending or forwarding a HelloRetryRequest, the client-facing server does not repeat the steps in Section 7.1 with the second ClientHelloOuter. Instead, it continues with the ECHConfig selection from the first ClientHelloOuter as follows:

If the client-facing server accepted ECH, it checks the second ClientHelloOuter also contains the "encrypted_client_hello" extension. If not, it MUST abort the handshake with a "missing_extension" alert. Otherwise, it checks that ECHClientHello.cipher_suite and ECHClientHello.config_id are unchanged, and that ECHClientHello.enc is empty. If not, it MUST abort the handshake with an "illegal_parameter" alert.

Finally, it decrypts the new ECHClientHello.payload as a second message with the previous HPKE context:

```
EncodedClientHelloInner = context.Open(ClientHelloOuterAAD,  
                                       ECHClientHello.payload)
```

ClientHelloOuterAAD is computed as described in Section 5.2, but using the second ClientHelloOuter. If decryption fails, the client-facing server MUST abort the handshake with a "decrypt_error" alert. Otherwise, it reconstructs the second ClientHelloInner from the new EncodedClientHelloInner as described in Section 5.1, using the second ClientHelloOuter for any referenced extensions.

The client-facing server then forwards the resulting ClientHelloInner to the backend server. It forwards all subsequent TLS messages between the client and backend server unmodified.

If the client-facing server rejected ECH, or if the first ClientHello did not include an "encrypted_client_hello" extension, the client-facing server proceeds with the connection as usual. The server does not decrypt the second ClientHello's ECHClientHello.payload value, if there is one. Moreover, if the server is configured with any

ECHConfigs, it MUST include the "encrypted_client_hello" extension in its EncryptedExtensions with the "retry_configs" field set to one or more ECHConfig structures with up-to-date keys, as described in Section 7.1.

Note that a client-facing server that forwards the first ClientHello cannot include its own "cookie" extension if the backend server sends a HelloRetryRequest. This means that the client-facing server either needs to maintain state for such a connection or it needs to coordinate with the backend server to include any information it requires to process the second ClientHello.

7.2. Backend Server

Upon receipt of an "encrypted_client_hello" extension of type inner in a ClientHello, if the backend server negotiates TLS 1.3 or higher, then it MUST confirm ECH acceptance to the client by computing its ServerHello as described here.

The backend server embeds in ServerHello.random a string derived from the inner handshake. It begins by computing its ServerHello as usual, except the last 8 bytes of ServerHello.random are set to zero. It then computes the transcript hash for ClientHelloInner up to and including the modified ServerHello, as described in [RFC8446], Section 4.4.1. Let transcript_ech_conf denote the output. Finally, the backend server overwrites the last 8 bytes of the ServerHello.random with the following string:

```
accept_confirmation = HKDF-Expand-Label(  
    HKDF-Extract(0, ClientHelloInner.random),  
    "ech accept confirmation",  
    transcript_ech_conf,  
    8)
```

where HKDF-Expand-Label is defined in [RFC8446], Section 7.1, "0" indicates a string of Hash.length bytes set to zero, and Hash is the hash function used to compute the transcript hash.

The backend server MUST NOT perform this operation if it negotiated TLS 1.2 or below. Note that doing so would overwrite the downgrade signal for TLS 1.3 (see [RFC8446], Section 4.1.3).

7.2.1. Sending HelloRetryRequest

When the backend server sends HelloRetryRequest in response to the ClientHello, it similarly confirms ECH acceptance by adding a confirmation signal to its HelloRetryRequest. But instead of embedding the signal in the HelloRetryRequest.random (the value of which is specified by [RFC8446]), it sends the signal in an extension.

The backend server begins by computing HelloRetryRequest as usual, except that it also contains an "encrypted_client_hello" extension with a payload of 8 zero bytes. It then computes the transcript hash for the first ClientHelloInner, denoted ClientHelloInner1, up to and including the modified HelloRetryRequest. Let transcript_hrr_ech_conf denote the output. Finally, the backend server overwrites the payload of the "encrypted_client_hello" extension with the following string:

```
hrr_accept_confirmation = HKDF-Expand-Label(  
    HKDF-Extract(0, ClientHelloInner1.random),  
    "hrr ech accept confirmation",  
    transcript_hrr_ech_conf,  
    8)
```

In the subsequent ServerHello message, the backend server sends the accept_confirmation value as described in Section 7.2.

8. Compatibility Issues

Unlike most TLS extensions, placing the SNI value in an ECH extension is not interoperable with existing servers, which expect the value in the existing plaintext extension. Thus server operators SHOULD ensure servers understand a given set of ECH keys before advertising them. Additionally, servers SHOULD retain support for any previously-advertised keys for the duration of their validity.

However, in more complex deployment scenarios, this may be difficult to fully guarantee. Thus this protocol was designed to be robust in case of inconsistencies between systems that advertise ECH keys and servers, at the cost of extra round-trips due to a retry. Two specific scenarios are detailed below.

8.1. Misconfiguration and Deployment Concerns

It is possible for ECH advertisements and servers to become inconsistent. This may occur, for instance, from DNS misconfiguration, caching issues, or an incomplete rollout in a multi-server deployment. This may also occur if a server loses its ECH keys, or if a deployment of ECH must be rolled back on the server.

The retry mechanism repairs inconsistencies, provided the server is authoritative for the public name. If server and advertised keys mismatch, the server will reject ECH and respond with "retry_configs". If the server does not understand the "encrypted_client_hello" extension at all, it will ignore it as required by Section 4.1.2 of [RFC8446]. Provided the server can present a certificate valid for the public name, the client can safely retry with updated settings, as described in Section 6.1.6.

Unless ECH is disabled as a result of successfully establishing a connection to the public name, the client **MUST NOT** fall back to using unencrypted ClientHellos, as this allows a network attacker to disclose the contents of this ClientHello, including the SNI. It **MAY** attempt to use another server from the DNS results, if one is provided.

8.2. Middleboxes

When connecting through a TLS-terminating proxy that does not support this extension, [RFC8446], Section 9.3 requires the proxy still act as a conforming TLS client and server. The proxy must ignore unknown parameters, and generate its own ClientHello containing only parameters it understands. Thus, when presenting a certificate to the client or sending a ClientHello to the server, the proxy will act as if connecting to the public name, without echoing the "encrypted_client_hello" extension.

Depending on whether the client is configured to accept the proxy's certificate as authoritative for the public name, this may trigger the retry logic described in Section 6.1.6 or result in a connection failure. A proxy which is not authoritative for the public name cannot forge a signal to disable ECH.

9. Compliance Requirements

In the absence of an application profile standard specifying otherwise, a compliant ECH application **MUST** implement the following HPKE cipher suite:

- * KEM: DHKEM(X25519, HKDF-SHA256) (see [I-D.irtf-cfrg-hpke], Section 7.1)
- * KDF: HKDF-SHA256 (see [I-D.irtf-cfrg-hpke], Section 7.2)
- * AEAD: AES-128-GCM (see [I-D.irtf-cfrg-hpke], Section 7.3)

10. Security Considerations

10.1. Security and Privacy Goals

ECH considers two types of attackers: passive and active. Passive attackers can read packets from the network, but they cannot perform any sort of active behavior such as probing servers or querying DNS. A middlebox that filters based on plaintext packet contents is one example of a passive attacker. In contrast, active attackers can also write packets into the network for malicious purposes, such as interfering with existing connections, probing servers, and querying DNS. In short, an active attacker corresponds to the conventional threat model for TLS 1.3 [RFC8446].

Given these types of attackers, the primary goals of ECH are as follows.

1. Use of ECH does not weaken the security properties of TLS without ECH.
2. TLS connection establishment to a host with a specific ECHConfig and TLS configuration is indistinguishable from a connection to any other host with the same ECHConfig and TLS configuration. (The set of hosts which share the same ECHConfig and TLS configuration is referred to as the anonymity set.)

Client-facing server configuration determines the size of the anonymity set. For example, if a client-facing server uses distinct ECHConfig values for each host, then each anonymity set has size $k = 1$. Client-facing servers SHOULD deploy ECH in such a way so as to maximize the size of the anonymity set where possible. This means client-facing servers should use the same ECHConfig for as many hosts as possible. An attacker can distinguish two hosts that have different ECHConfig values based on the ECHClientHello.config_id value. This also means public information in a TLS handshake should be consistent across hosts. For example, if a client-facing server services many backend origin hosts, only one of which supports some cipher suite, it may be possible to identify that host based on the contents of unencrypted handshake messages.

Beyond these primary security and privacy goals, ECH also aims to hide, to some extent, the fact that it is being used at all. Specifically, the GREASE ECH extension described in Section 6.2 does not change the security properties of the TLS handshake at all. Its goal is to provide "cover" for the real ECH protocol (Section 6.1), as a means of addressing the "do not stick out" requirements of [RFC8744]. See Section 10.9.4 for details.

10.2. Unauthenticated and Plaintext DNS

In comparison to [I-D.kazuho-protected-sni], wherein DNS Resource Records are signed via a server private key, ECH records have no authenticity or provenance information. This means that any attacker which can inject DNS responses or poison DNS caches, which is a common scenario in client access networks, can supply clients with fake ECH records (so that the client encrypts data to them) or strip the ECH record from the response. However, in the face of an attacker that controls DNS, no encryption scheme can work because the attacker can replace the IP address, thus blocking client connections, or substitute a unique IP address which is 1:1 with the DNS name that was looked up (modulo DNS wildcards). Thus, allowing the ECH records in the clear does not make the situation significantly worse.

Clearly, DNSSEC (if the client validates and hard fails) is a defense against this form of attack, but DoH/DPRIVE are also defenses against DNS attacks by attackers on the local network, which is a common case where ClientHello and SNI encryption are desired. Moreover, as noted in the introduction, SNI encryption is less useful without encryption of DNS queries in transit via DoH or DPRIVE mechanisms.

10.3. Client Tracking

A malicious client-facing server could distribute unique, per-client ECHConfig structures as a way of tracking clients across subsequent connections. On-path adversaries which know about these unique keys could also track clients in this way by observing TLS connection attempts.

The cost of this type of attack scales linearly with the desired number of target clients. Moreover, DNS caching behavior makes targeting individual users for extended periods of time, e.g., using per-client ECHConfig structures delivered via HTTPS RRs with high TTLs, challenging. Clients can help mitigate this problem by flushing any DNS or ECHConfig state upon changing networks.

10.4. Ignored Configuration Identifiers and Trial Decryption

Ignoring configuration identifiers may be useful in scenarios where clients and client-facing servers do not want to reveal information about the client-facing server in the "encrypted_client_hello" extension. In such settings, clients send a randomly generated config_id in the ECHClientHello. Servers in these settings must perform trial decryption since they cannot identify the client's chosen ECH key using the config_id value. As a result, ignoring configuration identifiers may exacerbate DoS attacks. Specifically, an adversary may send malicious ClientHello messages, i.e., those which will not decrypt with any known ECH key, in order to force wasteful decryption. Servers that support this feature should, for example, implement some form of rate limiting mechanism to limit the potential damage caused by such attacks.

Unless specified by the application using (D)TLS or externally configured, implementations MUST NOT use this mode.

10.5. Outer ClientHello

Any information that the client includes in the ClientHelloOuter is visible to passive observers. The client SHOULD NOT send values in the ClientHelloOuter which would reveal a sensitive ClientHelloInner property, such as the true server name. It MAY send values associated with the public name in the ClientHelloOuter.

In particular, some extensions require the client send a server-name-specific value in the ClientHello. These values may reveal information about the true server name. For example, the "cached_info" ClientHello extension [RFC7924] can contain the hash of a previously observed server certificate. The client SHOULD NOT send values associated with the true server name in the ClientHelloOuter. It MAY send such values in the ClientHelloInner.

A client may also use different preferences in different contexts. For example, it may send a different ALPN lists to different servers or in different application contexts. A client that treats this context as sensitive SHOULD NOT send context-specific values in ClientHelloOuter.

Values which are independent of the true server name, or other information the client wishes to protect, MAY be included in ClientHelloOuter. If they match the corresponding ClientHelloInner, they MAY be compressed as described in Section 5.1. However, note the payload length reveals information about which extensions are compressed, so inner extensions which only sometimes match the corresponding outer extension SHOULD NOT be compressed.

Clients MAY include additional extensions in ClientHelloOuter to avoid signaling unusual behavior to passive observers, provided the choice of value and value itself are not sensitive. See Section 10.9.4.

10.6. Related Privacy Leaks

ECH requires encrypted DNS to be an effective privacy protection mechanism. However, verifying the server's identity from the Certificate message, particularly when using the X509 CertificateType, may result in additional network traffic that may reveal the server identity. Examples of this traffic may include requests for revocation information, such as OCSP or CRL traffic, or requests for repository information, such as authorityInformationAccess. It may also include implementation-specific traffic for additional information sources as part of verification.

Implementations SHOULD avoid leaking information that may identify the server. Even when sent over an encrypted transport, such requests may result in indirect exposure of the server's identity, such as indicating a specific CA or service being used. To mitigate this risk, servers SHOULD deliver such information in-band when possible, such as through the use of OCSP stapling, and clients SHOULD take steps to minimize or protect such requests during certificate validation.

Attacks that rely on non-ECH traffic to infer server identity in an ECH connection are out of scope for this document. For example, a client that connects to a particular host prior to ECH deployment may later resume a connection to that same host after ECH deployment. An adversary that observes this can deduce that the ECH-enabled connection was made to a host that the client previously connected to and which is within the same anonymity set.

10.7. Cookies

Section 4.2.2 of [RFC8446] defines a cookie value that servers may send in HelloRetryRequest for clients to echo in the second ClientHello. While ECH encrypts the cookie in the second ClientHelloInner, the backend server's HelloRetryRequest is unencrypted. This means differences in cookies between backend servers, such as lengths or cleartext components, may leak information about the server identity.

Backend servers in an anonymity set SHOULD NOT reveal information in the cookie which identifies the server. This may be done by handling HelloRetryRequest statefully, thus not sending cookies, or by using the same cookie construction for all backend servers.

Note that, if the cookie includes a key name, analogous to Section 4 of [RFC5077], this may leak information if different backend servers issue cookies with different key names at the time of the connection. In particular, if the deployment operates in Split Mode, the backend servers may not share cookie encryption keys. Backend servers may mitigate this by either handling key rotation with trial decryption, or coordinating to match key names.

10.8. Attacks Exploiting Acceptance Confirmation

To signal acceptance, the backend server overwrites 8 bytes of its ServerHello.random with a value derived from the ClientHelloInner.random. (See Section 7.2 for details.) This behavior increases the likelihood of the ServerHello.random colliding with the ServerHello.random of a previous session, potentially reducing the overall security of the protocol. However, the remaining 24 bytes provide enough entropy to ensure this is not a practical avenue of attack.

On the other hand, the probability that two 8-byte strings are the same is non-negligible. This poses a modest operational risk. Suppose the client-facing server terminates the connection (i.e., ECH is rejected or bypassed): if the last 8 bytes of its ServerHello.random coincide with the confirmation signal, then the client will incorrectly presume acceptance and proceed as if the backend server terminated the connection. However, the probability of a false positive occurring for a given connection is only 1 in 2^{64} . This value is smaller than the probability of network connection failures in practice.

Note that the same bytes of the ServerHello.random are used to implement downgrade protection for TLS 1.3 (see [RFC8446], Section 4.1.3). These mechanisms do not interfere because the backend server only signals ECH acceptance in TLS 1.3 or higher.

10.9. Comparison Against Criteria

[RFC8744] lists several requirements for SNI encryption. In this section, we re-iterate these requirements and assess the ECH design against them.

10.9.1. Mitigate Cut-and-Paste Attacks

Since servers process either ClientHelloInner or ClientHelloOuter, and because ClientHelloInner.random is encrypted, it is not possible for an attacker to "cut and paste" the ECH value in a different Client Hello and learn information from ClientHelloInner.

10.9.2. Avoid Widely Shared Secrets

This design depends upon DNS as a vehicle for semi-static public key distribution. Server operators may partition their private keys however they see fit provided each server behind an IP address has the corresponding private key to decrypt a key. Thus, when one ECH key is provided, sharing is optimally bound by the number of hosts that share an IP address. Server operators may further limit sharing by publishing different DNS records containing ECHConfig values with different keys using a short TTL.

10.9.3. Prevent SNI-Based Denial-of-Service Attacks

This design requires servers to decrypt ClientHello messages with ECHClientHello extensions carrying valid digests. Thus, it is possible for an attacker to force decryption operations on the server. This attack is bound by the number of valid TCP connections an attacker can open.

10.9.4. Do Not Stick Out

As a means of reducing the impact of network ossification, [RFC8744] recommends SNI-protection mechanisms be designed in such a way that network operators do not differentiate connections using the mechanism from connections not using the mechanism. To that end, ECH is designed to resemble a standard TLS handshake as much as possible. The most obvious difference is the extension itself: as long as middleboxes ignore it, as required by [RFC8446], the rest of the handshake is designed to look very much as usual.

The GREASE ECH protocol described in Section 6.2 provides a low-risk way to evaluate the deployability of ECH. It is designed to mimic the real ECH protocol (Section 6.1) without changing the security properties of the handshake. The underlying theory is that if GREASE ECH is deployable without triggering middlebox misbehavior, and real ECH looks enough like GREASE ECH, then ECH should be deployable as well. Thus, our strategy for mitigating network ossification is to deploy GREASE ECH widely enough to disincentivize differential treatment of the real ECH protocol by the network.

Ensuring that networks do not differentiate between real ECH and GREASE ECH may not be feasible for all implementations. While most middleboxes will not treat them differently, some operators may wish to block real ECH usage but allow GREASE ECH. This specification aims to provide a baseline security level that most deployments can achieve easily, while providing implementations enough flexibility to achieve stronger security where possible. Minimally, real ECH is designed to be indifferentiable from GREASE ECH for passive adversaries with following capabilities:

1. The attacker does not know the ECHConfigList used by the server.
2. The attacker keeps per-connection state only. In particular, it does not track endpoints across connections.
3. ECH and GREASE ECH are designed so that the following features do not vary: the code points of extensions negotiated in the clear; the length of messages; and the values of plaintext alert messages.

This leaves a variety of practical differentiators out-of-scope. including, though not limited to, the following:

1. the value of the configuration identifier;
2. the value of the outer SNI;
3. the TLS version negotiated, which may depend on ECH acceptance;
4. client authentication, which may depend on ECH acceptance; and
5. HRR issuance, which may depend on ECH acceptance.

These can be addressed with more sophisticated implementations, but some mitigations require coordination between the client and server. These mitigations are out-of-scope for this specification.

10.9.5. Maintain Forward Secrecy

This design is not forward secret because the server's ECH key is static. However, the window of exposure is bound by the key lifetime. It is RECOMMENDED that servers rotate keys frequently.

10.9.6. Enable Multi-party Security Contexts

This design permits servers operating in Split Mode to forward connections directly to backend origin servers. The client authenticates the identity of the backend origin server, thereby avoiding unnecessary MiTM attacks.

Conversely, assuming ECH records retrieved from DNS are authenticated, e.g., via DNSSEC or fetched from a trusted Recursive Resolver, spoofing a client-facing server operating in Split Mode is not possible. See Section 10.2 for more details regarding plaintext DNS.

Authenticating the ECHConfig structure naturally authenticates the included public name. This also authenticates any retry signals from the client-facing server because the client validates the server certificate against the public name before retrying.

10.9.7. Support Multiple Protocols

This design has no impact on application layer protocol negotiation. It may affect connection routing, server certificate selection, and client certificate verification. Thus, it is compatible with multiple application and transport protocols. By encrypting the entire ClientHello, this design additionally supports encrypting the ALPN extension.

10.10. Padding Policy

Variations in the length of the ClientHelloInner ciphertext could leak information about the corresponding plaintext. Section 6.1.3 describes a RECOMMENDED padding mechanism for clients aimed at reducing potential information leakage.

10.11. Active Attack Mitigations

This section describes the rationale for ECH properties and mechanics as defenses against active attacks. In all the attacks below, the attacker is on-path between the target client and server. The goal of the attacker is to learn private information about the inner ClientHello, such as the true SNI value.

10.11.1. Client Reaction Attack Mitigation

This attack uses the client's reaction to an incorrect certificate as an oracle. The attacker intercepts a legitimate ClientHello and replies with a ServerHello, Certificate, CertificateVerify, and Finished messages, wherein the Certificate message contains a "test" certificate for the domain name it wishes to query. If the client decrypted the Certificate and failed verification (or leaked information about its verification process by a timing side channel), the attacker learns that its test certificate name was incorrect. As an example, suppose the client's SNI value in its inner ClientHello is "example.com," and the attacker replied with a Certificate for "test.com". If the client produces a verification failure alert because of the mismatch faster than it would due to the Certificate signature validation, information about the name leaks. Note that the attacker can also withhold the CertificateVerify message. In that scenario, a client which first verifies the Certificate would then respond similarly and leak the same information.

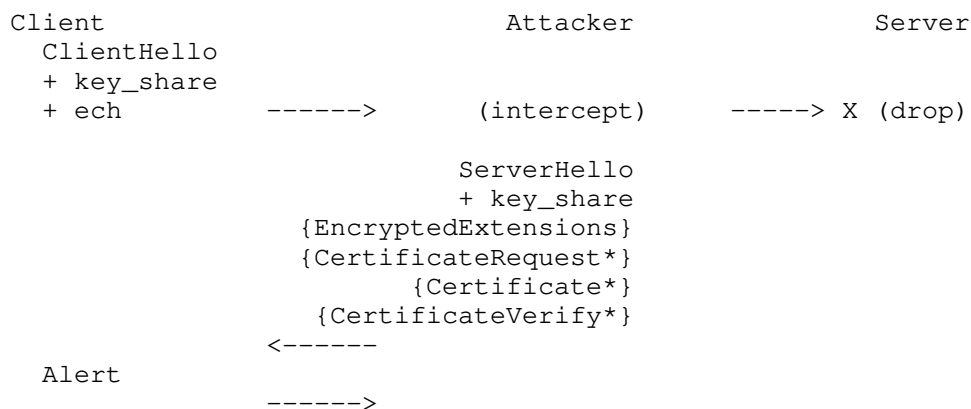


Figure 3: Client reaction attack

ClientHelloInner.random prevents this attack. In particular, since the attacker does not have access to this value, it cannot produce the right transcript and handshake keys needed for encrypting the Certificate message. Thus, the client will fail to decrypt the Certificate and abort the connection.

10.11.2. HelloRetryRequest Hijack Mitigation

This attack aims to exploit server HRR state management to recover information about a legitimate ClientHello using its own attacker-controlled ClientHello. To begin, the attacker intercepts and forwards a legitimate ClientHello with an "encrypted_client_hello" (ech) extension to the server, which triggers a legitimate HelloRetryRequest in return. Rather than forward the retry to the client, the attacker attempts to generate its own ClientHello in response based on the contents of the first ClientHello and HelloRetryRequest exchange with the result that the server encrypts the Certificate to the attacker. If the server used the SNI from the first ClientHello and the key share from the second (attacker-controlled) ClientHello, the Certificate produced would leak the client's chosen SNI to the attacker.

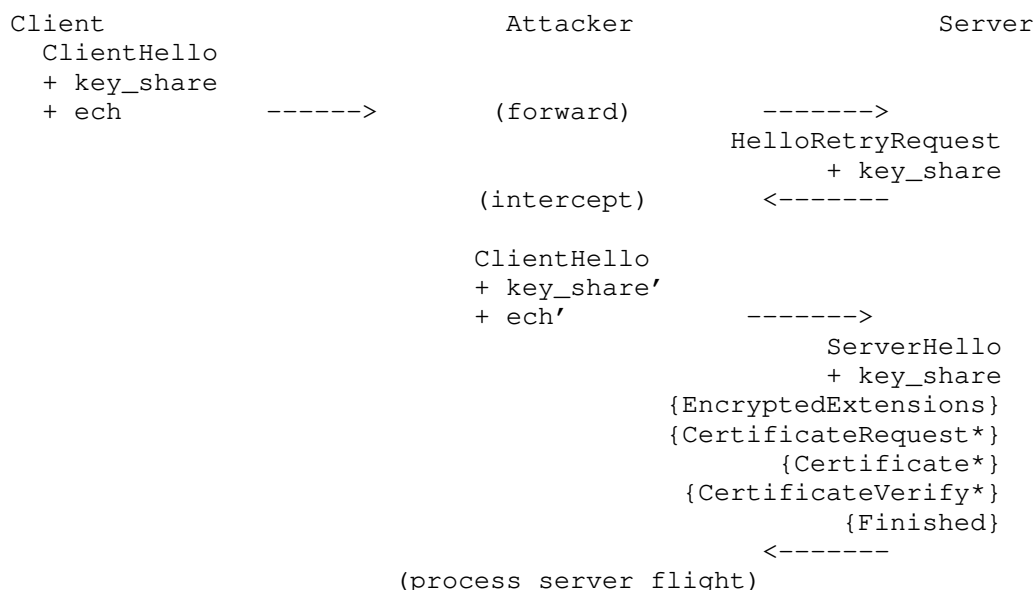


Figure 4: HelloRetryRequest hijack attack

This attack is mitigated by using the same HPKE context for both ClientHello messages. The attacker does not possess the context's keys, so it cannot generate a valid encryption of the second inner ClientHello.

If the attacker could manipulate the second ClientHello, it might be possible for the server to act as an oracle if it required parameters from the first ClientHello to match that of the second ClientHello. For example, imagine the client's original SNI value in the inner

ClientHello is "example.com", and the attacker's hijacked SNI value in its inner ClientHello is "test.com". A server which checks these for equality and changes behavior based on the result can be used as an oracle to learn the client's SNI.

10.11.3. ClientHello Malleability Mitigation

This attack aims to leak information about secret parts of the encrypted ClientHello by adding attacker-controlled parameters and observing the server's response. In particular, the compression mechanism described in Section 5.1 references parts of a potentially attacker-controlled ClientHelloOuter to construct ClientHelloInner, or a buggy server may incorrectly apply parameters from ClientHelloOuter to the handshake.

To begin, the attacker first interacts with a server to obtain a resumption ticket for a given test domain, such as "example.com". Later, upon receipt of a ClientHelloOuter, it modifies it such that the server will process the resumption ticket with ClientHelloInner. If the server only accepts resumption PSKs that match the server name, it will fail the PSK binder check with an alert when ClientHelloInner is for "example.com" but silently ignore the PSK and continue when ClientHelloInner is for any other name. This introduces an oracle for testing encrypted SNI values.

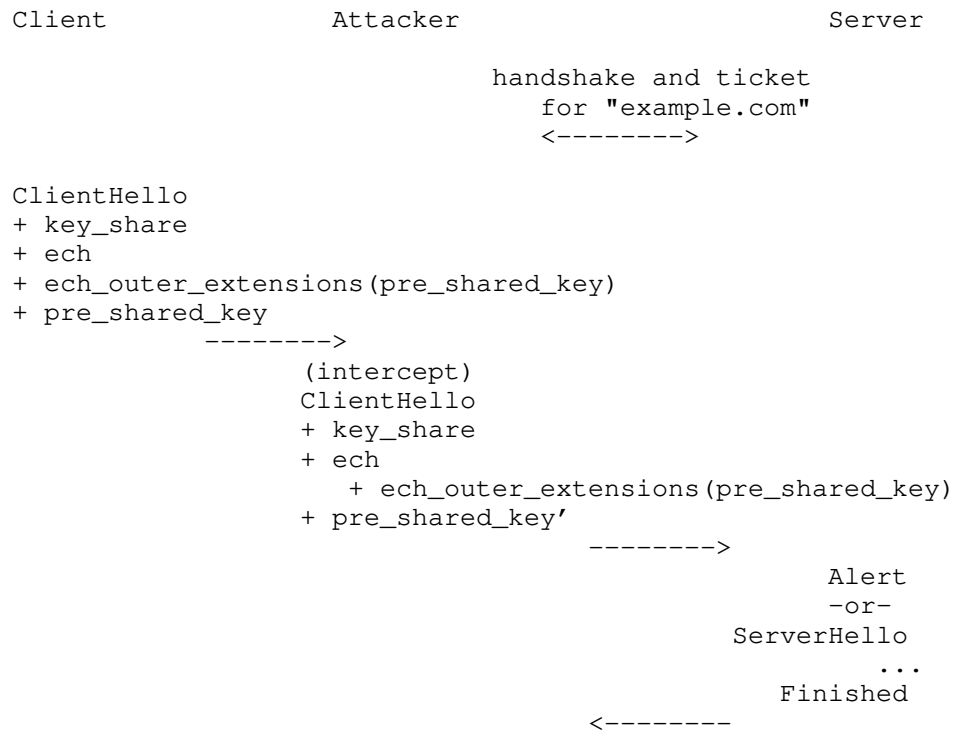


Figure 5: Message flow for malleable ClientHello

This attack may be generalized to any parameter which the server varies by server name, such as ALPN preferences.

ECH mitigates this attack by only negotiating TLS parameters from ClientHelloInner and authenticating all inputs to the ClientHelloInner (EncodedClientHelloInner and ClientHelloOuter) with the HPKE AEAD. See Section 5.2. An earlier iteration of this specification only encrypted and authenticated the "server_name" extension, which left the overall ClientHello vulnerable to an analogue of this attack.

10.11.4. ClientHelloInner Packet Amplification Mitigation

Client-facing servers must decompress EncodedClientHelloInners. A malicious attacker may craft a packet which takes excessive resources to decompress or may be much larger than the incoming packet:

- * If looking up a ClientHelloOuter extension takes time linear in the number of extensions, the overall decoding process would take $O(M*N)$ time, where M is the number of extensions in ClientHelloOuter and N is the size of OuterExtensions.
- * If the same ClientHelloOuter extension can be copied multiple times, an attacker could cause the client-facing server to construct a large ClientHelloInner by including a large extension in ClientHelloOuter, of length L , and an OuterExtensions list referencing N copies of that extension. The client-facing server would then use $O(N*L)$ memory in response to $O(N+L)$ bandwidth from the client. In split-mode, an $O(N*L)$ sized packet would then be transmitted to the backend server.

ECH mitigates this attack by requiring that OuterExtensions be referenced in order, that duplicate references be rejected, and by recommending that client-facing servers use a linear scan to perform decompression. These requirements are detailed in Section 5.1.

11. IANA Considerations

11.1. Update of the TLS ExtensionType Registry

IANA is requested to create the following entries in the existing registry for ExtensionType (defined in [RFC8446]):

1. encrypted_client_hello(0xfe0d), with "TLS 1.3" column values set to "CH, HRR, EE", and "Recommended" column set to "Yes".
2. ech_outer_extensions(0xfd00), with the "TLS 1.3" column values set to "", and "Recommended" column set to "Yes".

11.2. Update of the TLS Alert Registry

IANA is requested to create an entry, ech_required(121) in the existing registry for Alerts (defined in [RFC8446]), with the "DTLS-OK" column set to "Y".

12. ECHConfig Extension Guidance

Any future information or hints that influence ClientHelloOuter SHOULD be specified as ECHConfig extensions. This is primarily because the outer ClientHello exists only in support of ECH. Namely, it is both an envelope for the encrypted inner ClientHello and enabler for authenticated key mismatch signals (see Section 7). In contrast, the inner ClientHello is the true ClientHello used upon ECH negotiation.

13. References

13.1. Normative References

- [HTTPS-RR] Schwartz, B., Bishop, M., and E. Nygren, "Service binding and parameter specification via the DNS (DNS SVCB and HTTPS RRs)", Work in Progress, Internet-Draft, draft-ietf-dnsop-svcb-https-08, 12 October 2021, <<https://www.ietf.org/archive/id/draft-ietf-dnsop-svcb-https-08.txt>>.
- [I-D.ietf-tls-exported-authenticator] Sullivan, N., "Exported Authenticators in TLS", Work in Progress, Internet-Draft, draft-ietf-tls-exported-authenticator-14, 25 January 2021, <<https://www.ietf.org/archive/id/draft-ietf-tls-exported-authenticator-14.txt>>.
- [I-D.irtf-cfrg-hpke] Barnes, R. L., Bhargavan, K., Lipp, B., and C. A. Wood, "Hybrid Public Key Encryption", Work in Progress, Internet-Draft, draft-irtf-cfrg-hpke-12, 2 September 2021, <<https://www.ietf.org/archive/id/draft-irtf-cfrg-hpke-12.txt>>.
- [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/info/rfc2119>>.
- [RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, DOI 10.17487/RFC5890, August 2010, <<https://www.rfc-editor.org/info/rfc5890>>.
- [RFC7918] Langley, A., Modadugu, N., and B. Moeller, "Transport Layer Security (TLS) False Start", RFC 7918, DOI 10.17487/RFC7918, August 2016, <<https://www.rfc-editor.org/info/rfc7918>>.
- [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/info/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/info/rfc8446>>.

13.2. Informative References

- [I-D.kazuho-protected-sni]
Oku, K., "TLS Extensions for Protecting SNI", Work in Progress, Internet-Draft, draft-kazuho-protected-sni-00, 18 July 2017, <<https://www.ietf.org/archive/id/draft-kazuho-protected-sni-00.txt>>.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<https://www.rfc-editor.org/info/rfc3986>>.
- [RFC5077] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", RFC 5077, DOI 10.17487/RFC5077, January 2008, <<https://www.rfc-editor.org/info/rfc5077>>.
- [RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<https://www.rfc-editor.org/info/rfc7301>>.
- [RFC7858] Hu, Z., Zhu, L., Heidemann, J., Mankin, A., Wessels, D., and P. Hoffman, "Specification for DNS over Transport Layer Security (TLS)", RFC 7858, DOI 10.17487/RFC7858, May 2016, <<https://www.rfc-editor.org/info/rfc7858>>.
- [RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, July 2016, <<https://www.rfc-editor.org/info/rfc7924>>.
- [RFC8094] Reddy, T., Wing, D., and P. Patil, "DNS over Datagram Transport Layer Security (DTLS)", RFC 8094, DOI 10.17487/RFC8094, February 2017, <<https://www.rfc-editor.org/info/rfc8094>>.
- [RFC8484] Hoffman, P. and P. McManus, "DNS Queries over HTTPS (DoH)", RFC 8484, DOI 10.17487/RFC8484, October 2018, <<https://www.rfc-editor.org/info/rfc8484>>.
- [RFC8701] Benjamin, D., "Applying Generate Random Extensions And Sustain Extensibility (GREASE) to TLS Extensibility", RFC 8701, DOI 10.17487/RFC8701, January 2020, <<https://www.rfc-editor.org/info/rfc8701>>.

[RFC8744] Huitema, C., "Issues and Requirements for Server Name Identification (SNI) Encryption in TLS", RFC 8744, DOI 10.17487/RFC8744, July 2020, <<https://www.rfc-editor.org/info/rfc8744>>.

[WHATWG-IPV4] "URL Living Standard - IPv4 Parser", May 2021, <<https://url.spec.whatwg.org/#concept-ipv4-parser>>.

Appendix A. Alternative SNI Protection Designs

Alternative approaches to encrypted SNI may be implemented at the TLS or application layer. In this section we describe several alternatives and discuss drawbacks in comparison to the design in this document.

A.1. TLS-layer

A.1.1. TLS in Early Data

In this variant, TLS Client Hellos are tunneled within early data payloads belonging to outer TLS connections established with the client-facing server. This requires clients to have established a previous session --- and obtained PSKs --- with the server. The client-facing server decrypts early data payloads to uncover Client Hellos destined for the backend server, and forwards them onwards as necessary. Afterwards, all records to and from backend servers are forwarded by the client-facing server -- unmodified. This avoids double encryption of TLS records.

Problems with this approach are: (1) servers may not always be able to distinguish inner Client Hellos from legitimate application data, (2) nested 0-RTT data may not function correctly, (3) 0-RTT data may not be supported -- especially under DoS -- leading to availability concerns, and (4) clients must bootstrap tunnels (sessions), costing an additional round trip and potentially revealing the SNI during the initial connection. In contrast, encrypted SNI protects the SNI in a distinct Client Hello extension and neither abuses early data nor requires a bootstrapping connection.

A.1.2. Combined Tickets

In this variant, client-facing and backend servers coordinate to produce "combined tickets" that are consumable by both. Clients offer combined tickets to client-facing servers. The latter parse them to determine the correct backend server to which the Client Hello should be forwarded. This approach is problematic due to non-trivial coordination between client-facing and backend servers for

ticket construction and consumption. Moreover, it requires a bootstrapping step similar to that of the previous variant. In contrast, encrypted SNI requires no such coordination.

A.2. Application-layer

A.2.1. HTTP/2 CERTIFICATE Frames

In this variant, clients request secondary certificates with CERTIFICATE_REQUEST HTTP/2 frames after TLS connection completion. In response, servers supply certificates via TLS exported authenticators [I-D.ietf-tls-exported-authenticator] in CERTIFICATE frames. Clients use a generic SNI for the underlying client-facing server TLS connection. Problems with this approach include: (1) one additional round trip before peer authentication, (2) non-trivial application-layer dependencies and interaction, and (3) obtaining the generic SNI to bootstrap the connection. In contrast, encrypted SNI induces no additional round trip and operates below the application layer.

Appendix B. Linear-time Outer Extension Processing

The following procedure processes the "ech_outer_extensions" extension (see Section 5.1) in linear time, ensuring that each referenced extension in the ClientHelloOuter is included at most once:

1. Let I be zero and N be the number of extensions in ClientHelloOuter.
2. For each extension type, E, in OuterExtensions:
 - * If E is "encrypted_client_hello", abort the connection with an "illegal_parameter" alert and terminate this procedure.
 - * While I is less than N and the I-th extension of ClientHelloOuter does not have type E, increment I.
 - * If I is equal to N, abort the connection with an "illegal_parameter" alert and terminate this procedure.
 - * Otherwise, the I-th extension of ClientHelloOuter has type E. Copy it to the EncodedClientHelloInner and increment I.

Appendix C. Acknowledgements

This document draws extensively from ideas in [I-D.kazuho-protected-sni], but is a much more limited mechanism because it depends on the DNS for the protection of the ECH key. Richard Barnes, Christian Huitema, Patrick McManus, Matthew Prince, Nick Sullivan, Martin Thomson, and David Benjamin also provided important ideas and contributions.

Appendix D. Change Log

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

Issue and pull request numbers are listed with a leading octothorp.

D.1. Since draft-ietf-tls-esni-12

- * Abort on duplicate OuterExtensions (#514)
- * Improve EncodedClientHelloInner definition (#503)
- * Clarify retry configuration usage (#498)
- * Expand on config_id generation implications (#491)
- * Server-side acceptance signal extension GREASE (#481)
- * Refactor overview, client implementation, and middlebox sections (#480, #478, #475, #508)
- * Editorial improvements (#485, #488, #490, #495, #496, #499, #500, #501, #504, #505, #507, #510, #511)

D.2. Since draft-ietf-tls-esni-11

- * Move ClientHello padding to the encoding (#443)
- * Align codepoints (#464)
- * Relax OuterExtensions checks for alignment with RFC8446 (#467)
- * Clarify HRR acceptance and rejection logic (#470)
- * Editorial improvements (#468, #465, #462, #461)

D.3. Since draft-ietf-tls-esni-10

- * Make HRR confirmation and ECH acceptance explicit (#422, #423)
- * Relax computation of the acceptance signal (#420, #449)
- * Simplify ClientHelloOuterAAD generation (#438, #442)
- * Allow empty enc in ECHClientHello (#444)
- * Authenticate ECHClientHello extensions position in ClientHelloOuterAAD (#410)
- * Allow clients to send a dummy PSK and early_data in ClientHelloOuter when applicable (#414, #415)
- * Compress ECHConfigContents (#409)
- * Validate ECHConfig.contents.public_name (#413, #456)
- * Validate ClientHelloInner contents (#411)
- * Note split-mode challenges for HRR (#418)
- * Editorial improvements (#428, #432, #439, #445, #458, #455)

D.4. Since draft-ietf-tls-esni-09

- * Finalize HPKE dependency (#390)
- * Move from client-computed to server-chosen, one-byte config identifier (#376, #381)
- * Rename ECHConfigs to ECHConfigList (#391)
- * Clarify some security and privacy properties (#385, #383)

Authors' Addresses

Eric Rescorla
RTFM, Inc.

Email: ekr@rtfm.com

Kazuho Oku
Fastly

Email: kazuhooku@gmail.com

Nick Sullivan
Cloudflare

Email: nick@cloudflare.com

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net

Network Working Group
Internet-Draft
Intended status: Informational
Expires: 15 July 2022

D. Stebila
University of Waterloo
S. Fluhrer
Cisco Systems
S. Gueron
U. Haifa, Amazon Web Services
11 January 2022

Hybrid key exchange in TLS 1.3
draft-ietf-tls-hybrid-design-04

Abstract

Hybrid key exchange refers to using multiple key exchange algorithms simultaneously and combining the result with the goal of providing security even if all but one of the component algorithms is broken. It is motivated by transition to post-quantum cryptography. This document provides a construction for hybrid key exchange in the Transport Layer Security (TLS) protocol version 1.3.

Discussion of this work is encouraged to happen on the TLS IETF mailing list tls@ietf.org or on the GitHub repository which contains the draft: <https://github.com/dstebila/draft-ietf-tls-hybrid-design>.

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 15 July 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	2
1.1. Revision history	2
1.2. Terminology	4
1.3. Motivation for use of hybrid key exchange	5
1.4. Scope	6
1.5. Goals	6
2. Key encapsulation mechanisms	7
3. Construction for hybrid key exchange	8
3.1. Negotiation	8
3.2. Transmitting public keys and ciphertexts	9
3.3. Shared secret calculation	11
4. Discussion	12
5. IANA Considerations	12
6. Security Considerations	13
7. Acknowledgements	14
8. References	14
8.1. Normative References	14
8.2. Informative References	14
Appendix A. Related work	19
Authors' Addresses	20

1. Introduction

This document gives a construction for hybrid key exchange in TLS 1.3. The overall design approach is a simple, "concatenation"-based approach: each hybrid key exchange combination should be viewed as a single new key exchange method, negotiated and transmitted using the existing TLS 1.3 mechanisms.

This document does not propose specific post-quantum mechanisms; see Section 1.4 for more on the scope of this document.

1.1. Revision history

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

Earlier versions of this document categorized various design decisions one could make when implementing hybrid key exchange in TLS 1.3.

- * draft-ietf-tls-hybrid-design-03:
 - Some wording changes
 - Remove design considerations appendix
- * draft-ietf-tls-hybrid-design-03:
 - Remove specific code point examples and requested codepoint range for hybrid private use
 - Change "Open questions" to "Discussion"
 - Some wording changes
- * draft-ietf-tls-hybrid-design-02:
 - Bump to version -02 to avoid expiry
- * draft-ietf-tls-hybrid-design-01:
 - Forbid variable-length secret keys
 - Use fixed-length KEM public keys/ciphertexts
- * draft-ietf-tls-hybrid-design-00:
 - Allow `key_exchange` values from the same algorithm to be reused across multiple `KeyShareEntry` records in the same `ClientHello`.
- * draft-stebila-tls-hybrid-design-03:
 - Add requirement for KEMs to provide protection against key reuse.
 - Clarify FIPS-compliance of shared secret concatenation method.
- * draft-stebila-tls-hybrid-design-02:
 - Design considerations from draft-stebila-tls-hybrid-design-00 and draft-stebila-tls-hybrid-design-01 are moved to the appendix.
 - A single construction is given in the main body.

- * draft-stebila-tls-hybrid-design-01:
 - Add (Comb-KDF-1) and (Comb-KDF-2) options.
 - Add two candidate instantiations.
- * draft-stebila-tls-hybrid-design-00: Initial version.

1.2. Terminology

For the purposes of this document, it is helpful to be able to divide cryptographic algorithms into two classes:

- * "Traditional" algorithms: Algorithms which are widely deployed today, but which may be deprecated in the future. In the context of TLS 1.3 in 2019, examples of traditional key exchange algorithms include elliptic curve Diffie-Hellman using secp256r1 or x25519, or finite-field Diffie-Hellman.
- * "Next-generation" (or "next-gen") algorithms: Algorithms which are not yet widely deployed, but which may eventually be widely deployed. An additional facet of these algorithms may be that we have less confidence in their security due to them being relatively new or less studied. This includes "post-quantum" algorithms.

"Hybrid" key exchange, in this context, means the use of two (or more) key exchange algorithms based on different cryptographic assumptions, e.g., one traditional algorithm and one next-gen algorithm, with the purpose of the final session key being secure as long as at least one of the component key exchange algorithms remains unbroken. We use the term "component" algorithms to refer to the algorithms combined in a hybrid key exchange.

We note that some authors prefer the phrase "composite" to refer to the use of multiple algorithms, to distinguish from "hybrid public key encryption" in which a key encapsulation mechanism and data encapsulation mechanism are combined to create public key encryption.

The primary motivation of this document is preparing for post-quantum algorithms. However, it is possible that public key cryptography based on alternative mathematical constructions will be required independent of the advent of a quantum computer, for example because of a cryptanalytic breakthrough. As such we opt for the more generic term "next-generation" algorithms rather than exclusively "post-quantum" algorithms.

Note that TLS 1.3 uses the phrase "groups" to refer to key exchange algorithms - for example, the supported_groups extension - since all key exchange algorithms in TLS 1.3 are Diffie-Hellman-based. As a result, some parts of this document will refer to data structures or messages with the term "group" in them despite using a key exchange algorithm that is not Diffie-Hellman-based nor a group.

1.3. Motivation for use of hybrid key exchange

A hybrid key exchange algorithm allows early adopters eager for post-quantum security to have the potential of post-quantum security (possibly from a less-well-studied algorithm) while still retaining at least the security currently offered by traditional algorithms. They may even need to retain traditional algorithms due to regulatory constraints, for example FIPS compliance.

Ideally, one would not use hybrid key exchange: one would have confidence in a single algorithm and parameterization that will stand the test of time. However, this may not be the case in the face of quantum computers and cryptanalytic advances more generally.

Many (though not all) post-quantum algorithms currently under consideration are relatively new; they have not been subject to the same depth of study as RSA and finite-field or elliptic curve Diffie-Hellman, and thus the security community does not necessarily have as much confidence in their fundamental security, or the concrete security level of specific parameterizations.

Moreover, it is possible that after next-generation algorithms are defined, and for a period of time thereafter, conservative users may not have full confidence in some algorithms.

Some users may want to accelerate adoption of post-quantum cryptography due the threat of retroactive decryption: if a cryptographic assumption is broken due to the advent of a quantum computer or some other cryptanalytic breakthrough, confidentiality of information can be broken retroactively by any adversary who has passively recorded handshakes and encrypted communications. Hybrid key exchange enables potential security against retroactive decryption while not fully abandoning classical cryptosystems.

As such, there may be users for whom hybrid key exchange is an appropriate step prior to an eventual transition to next-generation algorithms.

1.4. Scope

This document focuses on hybrid ephemeral key exchange in TLS 1.3 [TLS13]. It intentionally does not address:

- * Selecting which next-generation algorithms to use in TLS 1.3, or algorithm identifiers or encoding mechanisms for next-generation algorithms. This selection will be based on the recommendations by the Crypto Forum Research Group (CFRG), which is currently waiting for the results of the NIST Post-Quantum Cryptography Standardization Project [NIST].
- * Authentication using next-generation algorithms. While quantum computers could retroactively decrypt previous sessions, session authentication cannot be retroactively broken.

1.5. Goals

The primary goal of a hybrid key exchange mechanism is to facilitate the establishment of a shared secret which remains secure as long as as one of the component key exchange mechanisms remains unbroken.

In addition to the primary cryptographic goal, there may be several additional goals in the context of TLS 1.3:

- * ***Backwards compatibility:** Clients and servers who are "hybrid-aware", i.e., compliant with whatever hybrid key exchange standard is developed for TLS, should remain compatible with endpoints and middle-boxes that are not hybrid-aware. The three scenarios to consider are:
 1. Hybrid-aware client, hybrid-aware server: These parties should establish a hybrid shared secret.
 2. Hybrid-aware client, non-hybrid-aware server: These parties should establish a traditional shared secret (assuming the hybrid-aware client is willing to downgrade to traditional-only).
 3. Non-hybrid-aware client, hybrid-aware server: These parties should establish a traditional shared secret (assuming the hybrid-aware server is willing to downgrade to traditional-only).

Ideally backwards compatibility should be achieved without extra round trips and without sending duplicate information; see below.

- * ***High performance:** Use of hybrid key exchange should not be prohibitively expensive in terms of computational performance. In general this will depend on the performance characteristics of the specific cryptographic algorithms used, and as such is outside the scope of this document. See [PST] for preliminary results about performance characteristics.
- * ***Low latency:** Use of hybrid key exchange should not substantially increase the latency experienced to establish a connection. Factors affecting this may include the following.
 - The computational performance characteristics of the specific algorithms used. See above.
 - The size of messages to be transmitted. Public key and ciphertext sizes for post-quantum algorithms range from hundreds of bytes to over one hundred kilobytes, so this impact can be substantial. See [PST] for preliminary results in a laboratory setting, and [LANGLEY] for preliminary results on more realistic networks.
 - Additional round trips added to the protocol. See below.
- * ***No extra round trips:** Attempting to negotiate hybrid key exchange should not lead to extra round trips in any of the three hybrid-aware/non-hybrid-aware scenarios listed above.
- * ***Minimal duplicate information:** Attempting to negotiate hybrid key exchange should not mean having to send multiple public keys of the same type.

2. Key encapsulation mechanisms

This document models key agreement as key encapsulation mechanisms (KEMs), which consist of three algorithms:

- * **KeyGen()** -> (pk, sk): A probabilistic key generation algorithm, which generates a public key pk and a secret key sk.
- * **Encaps(pk)** -> (ct, ss): A probabilistic encapsulation algorithm, which takes as input a public key pk and outputs a ciphertext ct and shared secret ss.
- * **Decaps(sk, ct)** -> ss: A decapsulation algorithm, which takes as input a secret key sk and ciphertext ct and outputs a shared secret ss, or in some cases a distinguished error value.

The main security property for KEMs is indistinguishability under adaptive chosen ciphertext attack (IND-CCA2), which means that shared secret values should be indistinguishable from random strings even given the ability to have other arbitrary ciphertexts decapsulated. IND-CCA2 corresponds to security against an active attacker, and the public key / secret key pair can be treated as a long-term key or reused. A common design pattern for obtaining security under key reuse is to apply the Fujisaki-Okamoto (FO) transform [FO] or a variant thereof [HHK].

A weaker security notion is indistinguishability under chosen plaintext attack (IND-CPA), which means that the shared secret values should be indistinguishable from random strings given a copy of the public key. IND-CPA roughly corresponds to security against a passive attacker, and sometimes corresponds to one-time key exchange.

Key exchange in TLS 1.3 is phrased in terms of Diffie-Hellman key exchange in a group. DH key exchange can be modeled as a KEM, with KeyGen corresponding to selecting an exponent x as the secret key and computing the public key g^x ; encapsulation corresponding to selecting an exponent y , computing the ciphertext g^y and the shared secret $g^{(xy)}$, and decapsulation as computing the shared secret $g^{(xy)}$. See [I-D.irtf-cfrg-hpke] for more details of such Diffie-Hellman-based key encapsulation mechanisms.

TLS 1.3 does not require that ephemeral public keys be used only in a single key exchange session; some implementations may reuse them, at the cost of limited forward secrecy. As a result, any KEM used in the manner described in this document MUST explicitly be designed to be secure in the event that the public key is reused, such as achieving IND-CCA2 security or having a transform like the Fujisaki-Okamoto transform [FO] [HHK] applied. While it is recommended that implementations avoid reuse of KEM public keys, implementations that do reuse KEM public keys MUST ensure that the number of reuses of a KEM public key abides by any bounds in the specification of the KEM or subsequent security analyses. Implementations MUST NOT reuse randomness in the generation of KEM ciphertexts.

3. Construction for hybrid key exchange

3.1. Negotiation

Each particular combination of algorithms in a hybrid key exchange will be represented as a NamedGroup and sent in the supported_groups extension. No internal structure or grammar is implied or required in the value of the identifier; they are simply opaque identifiers.

Each value representing a hybrid key exchange will correspond to an ordered pair of two algorithms. For example, a future document could specify that one codepoint corresponds to secp256r1+PQALG1, and another corresponds to x25519+PQALG1. (We note that this is independent from future documents standardizing solely post-quantum key exchange methods, which would have to be assigned their own identifier.)

Specific values shall be standardized by IANA in the TLS Supported Groups registry.

```
enum {  
  
    /* Elliptic Curve Groups (ECDHE) */  
    secp256r1(0x0017), secp384r1(0x0018), secp521r1(0x0019),  
    x25519(0x001D), x448(0x001E),  
  
    /* Finite Field Groups (DHE) */  
    ffdhe2048(0x0100), ffdhe3072(0x0101), ffdhe4096(0x0102),  
    ffdhe6144(0x0103), ffdhe8192(0x0104),  
  
    /* Hybrid Key Exchange Methods */  
    TBD(0xTBD), ...,  
  
    /* Reserved Code Points */  
    ffdhe_private_use(0x01FC..0x01FF),  
    ecdhe_private_use(0xFE00..0xFEFF),  
    (0xFFFF)  
} NamedGroup;
```

3.2. Transmitting public keys and ciphertexts

We take the relatively simple "concatenation approach": the messages from the two algorithms being hybridized will be concatenated together and transmitted as a single value, to avoid having to change existing data structures. The values are directly concatenated, without any additional encoding or length fields; this assumes that the representation and length of elements is fixed once the algorithm is fixed. If concatenation were to be used with values that are not fixed-length, a length prefix or other unambiguous encoding must be used to ensure that the composition of the two values is injective and requires a mechanism different from that specified in this document.

Recall that in TLS 1.3 a KEM public key or KEM ciphertext is represented as a KeyShareEntry:

```
struct {  
    NamedGroup group;  
    opaque key_exchange<1..2^16-1>;  
} KeyShareEntry;
```

These are transmitted in the extension_data fields of KeyShareClientHello and KeyShareServerHello extensions:

```
struct {  
    KeyShareEntry client_shares<0..2^16-1>;  
} KeyShareClientHello;  
  
struct {  
    KeyShareEntry server_share;  
} KeyShareServerHello;
```

The client's shares are listed in descending order of client preference; the server selects one algorithm and sends its corresponding share.

For a hybrid key exchange, the key_exchange field of a KeyShareEntry is the concatenation of the key_exchange field for each of the constituent algorithms. The order of shares in the concatenation is the same as the order of algorithms indicated in the definition of the NamedGroup.

For the client's share, the key_exchange value contains the concatenation of the pk outputs of the corresponding KEMs' KeyGen algorithms, if that algorithm corresponds to a KEM; or the (EC)DH ephemeral key share, if that algorithm corresponds to an (EC)DH group. For the server's share, the key_exchange value contains concatenation of the ct outputs of the corresponding KEMs' Encaps algorithms, if that algorithm corresponds to a KEM; or the (EC)DH ephemeral key share, if that algorithm corresponds to an (EC)DH group.

[TLS13] requires that ``The key_exchange values for each KeyShareEntry MUST be generated independently.'' In the context of this document, since the same algorithm may appear in multiple named groups, we relax the above requirement to allow the same key_exchange value for the same algorithm to be reused in multiple KeyShareEntry records sent in within the same ClientHello. However, key_exchange values for different algorithms MUST be generated independently.

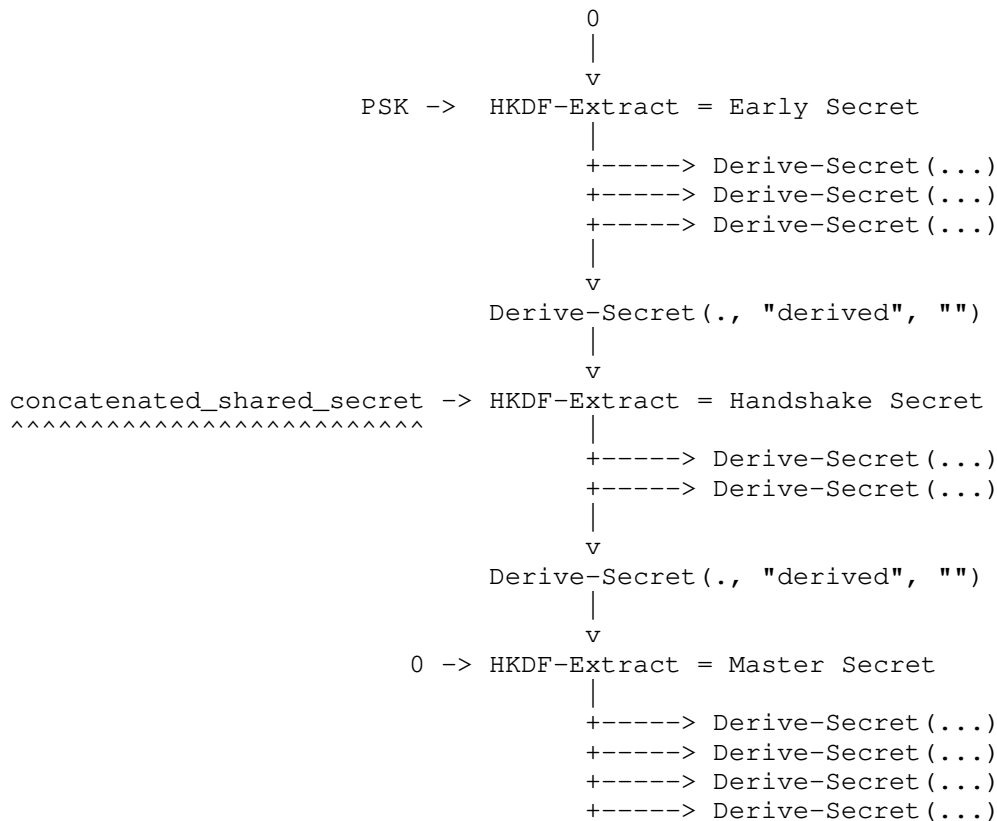
3.3. Shared secret calculation

Here we also take a simple "concatenation approach": the two shared secrets are concatenated together and used as the shared secret in the existing TLS 1.3 key schedule. Again, we do not add any additional structure (length fields) in the concatenation procedure: among all Round 3 finalists and alternate candidates, once the algorithm and variant are specified, the shared secret output length is fixed.

In other words, the shared secret is calculated as

```
concatenated_shared_secret = shared_secret_1 || shared_secret_2
```

and inserted into the TLS 1.3 key schedule in place of the (EC)DHE shared secret:



FIPS-compliance of shared secret concatenation. [NIST-SP-800-56C] or [NIST-SP-800-135] give NIST recommendations for key derivation methods in key exchange protocols. Some hybrid combinations may combine the shared secret from a NIST-approved algorithm (e.g., ECDH using the nistp256/secp256r1 curve) with a shared secret from a non-approved algorithm (e.g., post-quantum). [NIST-SP-800-56C] lists simple concatenation as an approved method for generation of a hybrid shared secret in which one of the constituent shared secret is from an approved method.

4. Discussion

Larger public keys and/or ciphertexts. The HybridKeyExchange struct in Section 3.2 limits public keys and ciphertexts to $2^{16}-1$ bytes; this is bounded by the same $(2^{16}-1)$ -byte limit on the key_exchange field in the KeyShareEntry struct. Some post-quantum KEMs have larger public keys and/or ciphertexts; for example, Classic McEliece's smallest parameter set has public key size 261,120 bytes. Hence this draft can not accommodate all current NIST Round 3 candidates.

Duplication of key shares. Concatenation of public keys in the HybridKeyExchange struct as described in Section 3.2 can result in sending duplicate key shares. For example, if a client wanted to offer support for two combinations, say "secp256r1+sikep503" and "x25519+sikep503", it would end up sending two sikep503 public keys, since the KeyShareEntry for each combination contains its own copy of a sikep503 key. This duplication may be more problematic for post-quantum algorithms which have larger public keys.

Failures. Some post-quantum key exchange algorithms have non-zero probability of failure, meaning two honest parties may derive different shared secrets. This would cause a handshake failure. All current NIST Round 3 candidates have either 0 or cryptographically small failure rate; if other algorithms are used, implementers should be aware of the potential of handshake failure. Clients can retry if a failure is encountered.

5. IANA Considerations

Identifiers for specific key exchange algorithm combinations will be defined in later documents.

6. Security Considerations

The shared secrets computed in the hybrid key exchange should be computed in a way that achieves the "hybrid" property: the resulting secret is secure as long as at least one of the component key exchange algorithms is unbroken. See [GIACON] and [BINDEL] for an investigation of these issues. Under the assumption that shared secrets are fixed length once the combination is fixed, the construction from Section 3.3 corresponds to the dual-PRF combiner of [BINDEL] which is shown to preserve security under the assumption that the hash function is a dual-PRF.

As noted in Section 2, KEMs used in the manner described in this document MUST explicitly be designed to be secure in the event that the public key is reused, such as achieving IND-CCA2 security or having a transform like the Fujisaki-Okamoto transform applied. Some IND-CPA-secure post-quantum KEMs (i.e., without countermeasures such as the FO transform) are completely insecure under public key reuse; for example, some lattice-based IND-CPA-secure KEMs are vulnerable to attacks that recover the private key after just a few thousand samples [FLUHRER].

Public keys, ciphertexts, and secrets should be constant length.
This document assumes that the length of each public key, ciphertext, and shared secret is fixed once the algorithm is fixed. This is the case for all Round 3 finalists and alternate candidates.

Note that variable-length secrets are, generally speaking, dangerous. In particular, when using key material of variable length and processing it using hash functions, a timing side channel may arise. In broad terms, when the secret is longer, the hash function may need to process more blocks internally. In some unfortunate circumstances, this has led to timing attacks, e.g. the Lucky Thirteen [LUCKY13] and Raccoon [RACCOON] attacks.

Furthermore, [AVIRAM] identified a risk of using variable-length secrets when the hash function used in the key derivation function is no longer collision-resistant.

Therefore, this specification MUST only be used with algorithms which have fixed-length shared secrets (after the variant has been fixed by the algorithm identifier in the NamedGroup negotiation in Section 3.1).

7. Acknowledgements

These ideas have grown from discussions with many colleagues, including Christopher Wood, Matt Campagna, Eric Crockett, authors of the various hybrid Internet-Drafts and implementations cited in this document, and members of the TLS working group. The immediate impetus for this document came from discussions with attendees at the Workshop on Post-Quantum Software in Mountain View, California, in January 2019. Daniel J. Bernstein and Tanja Lange commented on the risks of reuse of ephemeral public keys. Matt Campagna and the team at Amazon Web Services provided additional suggestions. Nimrod Aviram proposed restricting to fixed-length secrets.

8. References

8.1. Normative References

- [TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.

8.2. Informative References

- [AVIRAM] Nimrod Aviram, ., Benjamin Dowling, ., Ilan Komargodski, ., Kenny Paterson, ., Eyal Ronen, ., and . Eylon Yogev, "[TLS] Combining Secrets in Hybrid Key Exchange in TLS 1.3", 1 September 2021, <https://mailarchive.ietf.org/arch/msg/tls/F4SVeL2xbGPpPB2GW_GkBbD_a5M/>.
- [BCNS15] Bos, J., Costello, C., Naehrig, M., and D. Stebila, "Post-Quantum Key Exchange for the TLS Protocol from the Ring Learning with Errors Problem", 2015 IEEE Symposium on Security and Privacy, DOI 10.1109/sp.2015.40, May 2015, <<https://doi.org/10.1109/sp.2015.40>>.
- [BERNSTEIN] "Post-Quantum Cryptography", Springer Berlin Heidelberg book, DOI 10.1007/978-3-540-88702-7, 2009, <<https://doi.org/10.1007/978-3-540-88702-7>>.
- [BINDEL] Bindel, N., Brendel, J., Fischlin, M., Goncalves, B., and D. Stebila, "Hybrid Key Encapsulation Mechanisms and Authenticated Key Exchange", Post-Quantum Cryptography pp. 206-226, DOI 10.1007/978-3-030-25510-7_12, 2019, <https://doi.org/10.1007/978-3-030-25510-7_12>.

- [CAMPAGNA] Campagna, M. and E. Crockett, "Hybrid Post-Quantum Key Encapsulation Methods (PQ KEM) for Transport Layer Security 1.2 (TLS)", Work in Progress, Internet-Draft, draft-campagna-tls-bike-sike-hybrid-07, 2 September 2021, <<https://www.ietf.org/archive/id/draft-campagna-tls-bike-sike-hybrid-07.txt>>.
- [CECPQ1] Braithwaite, M., "Experimenting with Post-Quantum Cryptography", 7 July 2016, <<https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html>>.
- [CECPQ2] Langley, A., "CECPQ2", 12 December 2018, <<https://www.imperialviolet.org/2018/12/12/cecpq2.html>>.
- [DODIS] Dodis, Y. and J. Katz, "Chosen-Ciphertext Security of Multiple Encryption", Theory of Cryptography pp. 188-209, DOI 10.1007/978-3-540-30576-7_11, 2005, <https://doi.org/10.1007/978-3-540-30576-7_11>.
- [ETSI] Campagna, M., Ed. and . others, "Quantum safe cryptography and security: An introduction, benefits, enablers and challengers", ETSI White Paper No. 8 , June 2015, <<https://www.etsi.org/images/files/ETSIWhitePapers/QuantumSafeWhitepaper.pdf>>.
- [EVEN] Even, S. and O. Goldreich, "On the Power of Cascade Ciphers", Advances in Cryptology pp. 43-50, DOI 10.1007/978-1-4684-4730-9_4, 1984, <https://doi.org/10.1007/978-1-4684-4730-9_4>.
- [EXTERN-PSK] Housley, R., "TLS 1.3 Extension for Certificate-Based Authentication with an External Pre-Shared Key", RFC 8773, DOI 10.17487/RFC8773, March 2020, <<https://www.rfc-editor.org/info/rfc8773>>.
- [FLUHRER] Fluhrer, S., "Cryptanalysis of ring-LWE based key exchange with key share reuse", Cryptology ePrint Archive, Report 2016/085 , January 2016, <<https://eprint.iacr.org/2016/085>>.
- [FO] Fujisaki, E. and T. Okamoto, "Secure Integration of Asymmetric and Symmetric Encryption Schemes", Journal of Cryptology Vol. 26, pp. 80-101, DOI 10.1007/s00145-011-9114-1, December 2011, <<https://doi.org/10.1007/s00145-011-9114-1>>.

- [FRODO] Bos, J., Costello, C., Ducas, L., Mironov, I., Naehrig, M., Nikolaenko, V., Raghunathan, A., and D. Stebila, "Frodo: Take off the Ring! Practical, Quantum-Secure Key Exchange from LWE", Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, DOI 10.1145/2976749.2978425, October 2016, <<https://doi.org/10.1145/2976749.2978425>>.
- [GIACON] Giacon, F., Heuer, F., and B. Poettering, "KEM Combiners", Public-Key Cryptography - PKC 2018 pp. 190-218, DOI 10.1007/978-3-319-76578-5_7, 2018, <https://doi.org/10.1007/978-3-319-76578-5_7>.
- [HARNIK] Harnik, D., Kilian, J., Naor, M., Reingold, O., and A. Rosen, "On Robust Combiners for Oblivious Transfer and Other Primitives", Lecture Notes in Computer Science pp. 96-113, DOI 10.1007/11426639_6, 2005, <https://doi.org/10.1007/11426639_6>.
- [HHK] Hofheinz, D., Hovelmanns, K., and E. Kiltz, "A Modular Analysis of the Fujisaki-Okamoto Transformation", Theory of Cryptography pp. 341-371, DOI 10.1007/978-3-319-70500-2_12, 2017, <https://doi.org/10.1007/978-3-319-70500-2_12>.
- [HOFFMAN] Hoffman, P., "The Transition from Classical to Post-Quantum Cryptography", Work in Progress, Internet-Draft, draft-hoffman-c2pq-07, 26 May 2020, <<https://www.ietf.org/archive/id/draft-hoffman-c2pq-07.txt>>.
- [I-D.irtf-cfrg-hpke] Barnes, R. L., Bhargavan, K., Lipp, B., and C. A. Wood, "Hybrid Public Key Encryption", Work in Progress, Internet-Draft, draft-irtf-cfrg-hpke-12, 2 September 2021, <<https://www.ietf.org/archive/id/draft-irtf-cfrg-hpke-12.txt>>.
- [IKE-HYBRID] Tjhai, C., Tomlinson, M., Bartlett, G., Fluhrer, S., Geest, D. V., Garcia-Morchon, O., and V. Smyslov, "Framework to Integrate Post-quantum Key Exchanges into Internet Key Exchange Protocol Version 2 (IKEv2)", Work in Progress, Internet-Draft, draft-tjhai-ipsecme-hybrid-qske-ikev2-04, 9 July 2019, <<https://www.ietf.org/archive/id/draft-tjhai-ipsecme-hybrid-qske-ikev2-04.txt>>.

- [IKE-PSK] Fluhrer, S., Kampanakis, P., McGrew, D., and V. Smyslov, "Mixing Preshared Keys in the Internet Key Exchange Protocol Version 2 (IKEv2) for Post-quantum Security", RFC 8784, DOI 10.17487/RFC8784, June 2020, <<https://www.rfc-editor.org/info/rfc8784>>.
- [KIEFER] Kiefer, F. and K. Kwiatkowski, "Hybrid ECDHE-SIDH Key Exchange for TLS", Work in Progress, Internet-Draft, draft-kiefer-tls-ecdhe-sidh-00, 5 November 2018, <<https://www.ietf.org/archive/id/draft-kiefer-tls-ecdhe-sidh-00.txt>>.
- [LANGLEY] Langley, A., "Post-quantum confidentiality for TLS", 11 April 2018, <<https://www.imperialviolet.org/2018/04/11/pqconftls.html>>.
- [LUCKY13] Al Fardan, N.J. and K.G. Paterson, "Lucky Thirteen: Breaking the TLS and DTLS record protocols", n.d., <<https://ieeexplore.ieee.org/iel7/6547086/6547088/06547131.pdf>>.
- [NIELSEN] Nielsen, M.A. and I.L. Chuang, "Quantum Computation and Quantum Information", Cambridge University Press , 2000.
- [NIST] National Institute of Standards and Technology (NIST), "Post-Quantum Cryptography", n.d., <<https://www.nist.gov/pqcrypto>>.
- [NIST-SP-800-135]
National Institute of Standards and Technology (NIST), "Recommendation for Existing Application-Specific Key Derivation Functions", December 2011, <<https://doi.org/10.6028/NIST.SP.800-135r1>>.
- [NIST-SP-800-56C]
National Institute of Standards and Technology (NIST), "Recommendation for Key-Derivation Methods in Key-Establishment Schemes", August 2020, <<https://doi.org/10.6028/NIST.SP.800-56Cr2>>.
- [OQS-102] Open Quantum Safe Project, "OQS-OpenSSL-1-0-2_stable", November 2018, <https://github.com/open-quantum-safe/openssl/tree/OQS-OpenSSL_1_0_2-stable>.
- [OQS-111] Open Quantum Safe Project, "OQS-OpenSSL-1-1-1_stable", January 2022, <https://github.com/open-quantum-safe/openssl/tree/OQS-OpenSSL_1_1_1-stable>.

- [PST] Paquin, C., Stebila, D., and G. Tamvada, "Benchmarking Post-quantum Cryptography in TLS", Post-Quantum Cryptography pp. 72-91, DOI 10.1007/978-3-030-44223-1_5, 2020, <https://doi.org/10.1007/978-3-030-44223-1_5>.
- [RACCOON] Merget, R., Brinkmann, M., Aviram, N., Somorovsky, J., Mittmann, J., and J. Schwenk, "Raccoon Attack: Finding and Exploiting Most-Significant-Bit-Oracles in TLS-DH(E)", September 2020, <<https://raccoon-attack.com/>>.
- [S2N] Amazon Web Services, "Post-quantum TLS now supported in AWS KMS", 4 November 2019, <<https://aws.amazon.com/blogs/security/post-quantum-tls-now-supported-in-aws-kms/>>.
- [SCHANCK] Schanck, J. M. and D. Stebila, "A Transport Layer Security (TLS) Extension For Establishing An Additional Shared Secret", Work in Progress, Internet-Draft, draft-schanck-tls-additional-keyshare-00, 17 April 2017, <<https://www.ietf.org/archive/id/draft-schanck-tls-additional-keyshare-00.txt>>.
- [WHYTE12] Schanck, J. M., Whyte, W., and Z. Zhang, "Quantum-Safe Hybrid (QSH) Ciphersuite for Transport Layer Security (TLS) version 1.2", Work in Progress, Internet-Draft, draft-whyte-qsh-tls12-02, 22 July 2016, <<https://www.ietf.org/archive/id/draft-whyte-qsh-tls12-02.txt>>.
- [WHYTE13] Whyte, W., Zhang, Z., Fluhrer, S., and O. Garcia-Morchon, "Quantum-Safe Hybrid (QSH) Key Exchange for Transport Layer Security (TLS) version 1.3", Work in Progress, Internet-Draft, draft-whyte-qsh-tls13-06, 3 October 2017, <<https://www.ietf.org/archive/id/draft-whyte-qsh-tls13-06.txt>>.
- [XMSS] Huelsing, A., Butin, D., Gazdag, S., Rijneveld, J., and A. Mohaisen, "XMSS: eXtended Merkle Signature Scheme", RFC 8391, DOI 10.17487/RFC8391, May 2018, <<https://www.rfc-editor.org/info/rfc8391>>.
- [ZHANG] Zhang, R., Hanaoka, G., Shikata, J., and H. Imai, "On the Security of Multiple Encryption or CCA-security+CCA-security=CCA-security?", Public Key Cryptography - PKC 2004 pp. 360-374, DOI 10.1007/978-3-540-24632-9_26, 2004, <https://doi.org/10.1007/978-3-540-24632-9_26>.

Appendix A. Related work

Quantum computing and post-quantum cryptography in general are outside the scope of this document. For a general introduction to quantum computing, see a standard textbook such as [NIELSEN]. For an overview of post-quantum cryptography as of 2009, see [BERNSTEIN]. For the current status of the NIST Post-Quantum Cryptography Standardization Project, see [NIST]. For additional perspectives on the general transition from classical to post-quantum cryptography, see for example [ETSI] and [HOFFMAN], among others.

There have been several Internet-Drafts describing mechanisms for embedding post-quantum and/or hybrid key exchange in TLS:

- * Internet-Drafts for TLS 1.2: [WHYTE12], [CAMPAGNA]

- * Internet-Drafts for TLS 1.3: [KIEFER], [SCHANCK], [WHYTE13]

There have been several prototype implementations for post-quantum and/or hybrid key exchange in TLS:

- * Experimental implementations in TLS 1.2: [BCNS15], [CECPQ1], [FRODO], [OQS-102], [S2N]

- * Experimental implementations in TLS 1.3: [CECPQ2], [OQS-111], [PST]

These experimental implementations have taken an ad hoc approach and not attempted to implement one of the drafts listed above.

Unrelated to post-quantum but still related to the issue of combining multiple types of keying material in TLS is the use of pre-shared keys, especially the recent TLS working group document on including an external pre-shared key [EXTERN-PSK].

Considering other IETF standards, there is work on post-quantum preshared keys in IKEv2 [IKE-PSK] and a framework for hybrid key exchange in IKEv2 [IKE-HYBRID]. The XMSS hash-based signature scheme has been published as an informational RFC by the IRTF [XMSS].

In the academic literature, [EVEN] initiated the study of combining multiple symmetric encryption schemes; [ZHANG], [DODIS], and [HARNIK] examined combining multiple public key encryption schemes, and [HARNIK] coined the term "robust combiner" to refer to a compiler that constructs a hybrid scheme from individual schemes while preserving security properties. [GIACON] and [BINDEL] examined combining multiple key encapsulation mechanisms.

Authors' Addresses

Douglas Stebila
University of Waterloo

Email: dstebila@uwaterloo.ca

Scott Fluhrer
Cisco Systems

Email: sfluhrer@cisco.com

Shay Gueron
University of Haifa and Amazon Web Services

Email: shay.gueron@gmail.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: 7 January 2022

M. Thomson
Mozilla
6 July 2021

Secure Negotiation of Incompatible Protocols in TLS
draft-thomson-tls-snip-02

Abstract

An extension is defined for TLS that allows a client and server to detect an attempt to force the use of less-preferred application protocol even where protocol options are incompatible. This supplements application-layer protocol negotiation (ALPN), which allows choices between compatible protocols to be authenticated.

Discussion Venues

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

Discussion of this document takes place on the TLS Working Group mailing list (tls@ietf.org), which is archived at <https://mailarchive.ietf.org/arch/browse/tls/>.

Source for this draft and an issue tracker can be found at <https://github.com/martinthomson/snip>.

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 7 January 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Terminology	3
3. Incompatible Protocols and SVCB	4
4. Authenticating Incompatible Protocols	4
5. Incompatible Protocol Selection	5
6. Logical Servers	6
6.1. Validation Process	7
6.2. QUIC Version Negotiation	7
6.3. Alternative Services	7
7. Operational Considerations	8
8. Security Considerations	8
9. IANA Considerations	9
10. References	9
10.1. Normative References	9
10.2. Informative References	9
Appendix A. Acknowledgments	10
Appendix B. Defining Logical Servers	11
Author's Address	11

1. Introduction

With increased diversity in protocol choice, some applications are able to use one of several semantically-equivalent protocols to achieve their goals. This is particularly notable in HTTP where there are currently three distinct protocols: HTTP/1.1 [HTTP11], HTTP/2 [HTTP2], and HTTP/3 [HTTP3]. This is also true of protocols that support variants based on both TLS [TLS] and DTLS [DTLS].

For protocols that are mutually compatible, Application-Layer Protocol Negotiation (ALPN; [ALPN]) provides a secure way to negotiate protocol selection.

In ALPN, the client offers a list of options in a TLS ClientHello and the server chooses the option that it most prefers. A downgrade attack occurs where both client and server support a protocol that the server prefers more than the selected protocol. ALPN protects against this attack by ensuring that the server is aware of all options the client supports and including those options and the server choice under the integrity protection provided by the TLS handshake.

This downgrade protection functions because protocol negotiation is part of the TLS handshake. The introduction of semantically-equivalent protocols that use incompatible handshakes introduces new opportunities for downgrade attack. For instance, it is not possible to negotiate the use of HTTP/2 based on an attempt to connect using HTTP/3. The former relies on TCP, whereas the latter uses UDP. These protocols are therefore mutually incompatible.

This document defines an extension to TLS that allows clients to discover when servers support alternative protocols that are incompatible with the currently-selected TLS version. This might be used to avoid downgrade attack caused by interference in protocol discovery mechanisms.

This extension is motivated by the addition of new mechanisms, such as [SVCB]. SVCB enables the discovery of servers that support multiple different protocols, some of which are incompatible. The extension can also be used to authenticate protocol choices that are discovered by other means.

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.

Two protocols are considered "compatible" if it is possible to negotiate either using the same connection attempt. In comparison, protocols are "incompatible" if they require separate attempts to establish a connection.

3. Incompatible Protocols and SVCB

The SVCB record [SVCB] allows a client to learn about services associated with a domain name. This includes how to locate a server, along with supplementary information about the server, including protocols that the server supports. This allows a client to start using a protocol of their choice without added latency, as the lookup can be performed concurrently with other name resolution. The added cost of the additional DNS queries is minimal.

However, SVCB provides no protection against a downgrade attack between incompatible protocols. An attacker could remove DNS records for client-preferred protocols, leaving the client to believe that only less-preferred options are available. If those options are not compatible with the client-preferred option, the client will not know to attempt these. The client then only offers options compatible with the less-preferred options when attempting a TLS handshake. Even if a client were to inform the server that it supports a more preferred protocol, the server would not be able to act upon it.

Authenticating all of the information presented in SVCB records might provide clients with complete information about server support, but this is impractical for several reasons:

- * it is not possible to ensure that all server instances in a deployment have the same protocol configuration, as deployments for a single name routinely include multiple providers that cannot coordinate closely;
- * the ability to provide a subset of valid DNS records is integral to many strategies for managing servers; and
- * it is difficult to ensure that cached DNS records are synchronized with server state.

Overall, an authenticated TLS handshake is a better source of authoritative information about the protocols that are supported by servers.

4. Authenticating Incompatible Protocols

The `incompatible_protocols(TBD)` TLS extension provides clients with information about the incompatible protocols that are supported by the same logical server; see Section 6 for a definition of a logical server.

```
enum {  
    incompatible_protocols(TBD), (65535)  
} ExtensionType;
```

A client that supports the extension advertises an empty extension. In response, a server that supports this extension includes a list of application protocol identifiers. The "extension_data" field of the value server extension uses the "ProtocolName" type defined in [ALPN], which is repeated here. This syntax is shown in Figure 1.

```
opaque ProtocolName<1..2^8-1>;  
ProtocolName IncompatibleProtocol;  
  
struct {  
    select (Handshake.msg_type) {  
        case client_hello:  
            Empty;  
        case encrypted_extensions:  
            IncompatibleProtocol incompatible_protocols<3..2^16-1>;  
    };  
} IncompatibleProtocols;
```

Figure 1: TLS Syntax for incompatible_protocols Extension

This extension only applies to the ClientHello and EncryptedExtensions messages. An implementation that receives this extension in any other handshake message MUST send a fatal illegal_parameter alert.

A server deployment that supports multiple incompatible protocols MAY advertise all protocols that are supported by the same logical server. A server needs to ensure that protocols advertised in this fashion are available to the client.

A server MUST omit any compatible protocols from this extension. That is, any protocol that the server might be able to select, had the client offered the protocol in the application_layer_protocol_negotiation extension. In comparison, clients are expected to include all compatible protocols in the application_layer_protocol_negotiation extension.

5. Incompatible Protocol Selection

This document expands the definition of protocol negotiation to include both compatible and incompatible protocols and provide protection against downgrade for both types of selection. ALPN [ALPN] only considers compatible protocols: the client presents a set of compatible options and the server chooses its most preferred.

With an selection of protocols that includes incompatible options, the client makes a selection between incompatible options before making a connection attempt. Therefore, this design does not enable negotiation, it instead provides the client with information about other incompatible protocols that the server might support.

Detecting a potential downgrade between incompatible protocols does not automatically imply that a client abandon a connection attempt. It only provides the client with authenticated information about its options. What a client does with this information is left to client policy.

In brief:

- * For compatible protocols, the client offers all acceptable options and the server selects its most preferred
- * For incompatible protocols, information the server offers is authenticated and the client is able to act on that

For a protocol like HTTP/3, this might not result in the client choosing to use HTTP/3, even if HTTP/3 is preferred and the server indicates that a service endpoint supporting HTTP/3 is available. Blocking of UDP or QUIC is known to be widespread. As a result, clients might adopt a policy of tolerating a downgrade to a TCP-based protocol, even if HTTP/3 were preferred. However, as blocking of UDP is highly correlated by access network, clients that are able to establish HTTP/3 connections to some servers might choose to apply a stricter policy when a server that indicates HTTP/3 support is unreachable.

6. Logical Servers

The set of endpoints over which clients can assume availability of incompatible protocols is the set of endpoints that share an IP version, IP address, and port number with the TLS server that provides the incompatible_protocols extension.

This definition includes a port number that is independent of the protocol that is used. Any protocol that defines a port number is considered to be equivalent. In particular, incompatible protocols can be deployed to TCP, UDP, SCTP, or DCCP ports as long as the IP address and port number is the same.

This determination is made from the perspective of a client. This means that servers need to be aware of all instances that might answer to the same IP address and port; see Section 7.

6.1. Validation Process

The type of protocol authentication scope describes how a client might learn of all of the service endpoints that a server offers in that scope. If a client has attempted to discover service endpoints using the methods defined by the protocol authentication scope, receiving an `incompatible_protocols` extension from a server is a strong indication of a potential downgrade attack.

A client considers that a downgrade attack might have occurred if a server advertises that there are endpoints that support a protocol that the client prefers over the protocol that is currently in use.

In response to detecting a potential downgrade attack, a client might abandon the current connection attempt and report an error. A client that supports discovery of incompatible protocols, but chooses not to make a discovery attempt under normal conditions might instead not fail, but it could use what it learns as cause to initiate discovery.

6.2. QUIC Version Negotiation

QUIC enables the definition of incompatible protocols that share a port. This mechanism can be used to authenticate the choice of application protocol in QUIC. QUIC version negotiation [QUIC-VN] is used to authenticate the choice of QUIC version.

As there are two potentially competing sets of preferences, clients need to set preferences for QUIC version and application protocol that do not result in inconsistent outcomes. For example, if application protocol A exclusively uses QUIC version X and application protocol B exclusively uses QUIC version Y, setting a preference for both A and Y will lead to a failure condition that cannot be reconciled.

6.3. Alternative Services

It is possible to negotiate protocols based on an established connection without exposure to downgrade. The Alternative Services [ALTSVC] bootstrapping in HTTP/3 [HTTP3] does just that. Assuming that HTTP/2 or HTTP/1.1 are not vulnerable to attacks that would compromise integrity, a server can advertise the presence of an endpoint that supports HTTP/3.

Under these assumptions Alternative Services is secure, but it has performance trade-offs. A client could attempt the protocol it prefers most, but that comes at a risk that this protocol is not supported by a server. A client could implement a fallback, which might even be performed concurrently (see [HAPPY-EYEBALLS]), but this

costs time and resources. A client avoids these costs by attempting the protocol it believes to be most widely supported, though this comes with a performance penalty in cases where the most-preferred protocol is supported.

A client therefore choose to ignore incompatible protocols when attempting to use an alternative service.

7. Operational Considerations

By listing incompatible protocols a server needs reliable knowledge of the existence of these alternatives. This depends on some coordination of deployments. In particular, coordination is important if a load balancer distributes load for a single IP address to multiple server instances. Ensuring consistent configuration of servers could present operational difficulties as it requires that incompatible protocols are only listed when those protocols are deployed across all server instances.

Server deployments can choose not to provide information about incompatible protocols, which denies clients information about downgrade attacks but might avoid the operational complexity of providing accurate information.

During rollout of a new, incompatible protocol, until the deployment is stable and not at risk of being disabled, servers SHOULD NOT advertise the existence of the new protocol. Protocol deployments that are disabled, first need to be removed from the `incompatible_protocols` extension or there could be some loss of service. Though the `incompatible_protocols` extension only applies at the time of the TLS handshake, clients might take some time to act on the information. If an incompatible protocol is removed from deployment between when the client completes a handshake and when it acts, this could be treated as an error by the client.

If a server does not list incompatible protocols, clients cannot learn about other services and so cannot detect downgrade attacks against those protocols.

8. Security Considerations

This design depends on the integrity of the TLS handshake across all forms, including TLS [RFC8446], DTLS [DTLS], and QUIC [QUIC-TLS]. An attacker that can modify a TLS handshake in any one of these protocols can cause a client to believe that other options do not exist.

9. IANA Considerations

TODO: register the extension

10. References

10.1. Normative References

- [ALPN] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<https://www.rfc-editor.org/rfc/rfc7301>>.
- [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>>.

10.2. Informative References

- [ALTSVC] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <<https://www.rfc-editor.org/rfc/rfc7838>>.
- [DTLS] Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-dtls13-43, 30 April 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-tls-dtls13-43>>.
- [HAPPY-EYEBALLS] Wing, D. and A. Yourtchenko, "Happy Eyeballs: Success with Dual-Stack Hosts", RFC 6555, DOI 10.17487/RFC6555, April 2012, <<https://www.rfc-editor.org/rfc/rfc6555>>.
- [HTTP11] Fielding, R. T., Nottingham, M., and J. Reschke, "HTTP/1.1", Work in Progress, Internet-Draft, draft-ietf-httpbis-messaging-16, 27 May 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-messaging-16>>.

- [HTTP2] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/rfc/rfc7540>>.
- [HTTP3] Bishop, M., "Hypertext Transfer Protocol Version 3 (HTTP/3)", Work in Progress, Internet-Draft, draft-ietf-quic-http-34, 2 February 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-34>>.
- [QUIC-TLS] Thomson, M. and S. Turner, "Using TLS to Secure QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-tls-34, 14 January 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-quic-tls-34>>.
- [QUIC-VN] Schinazi, D. and E. Rescorla, "Compatible Version Negotiation for QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-version-negotiation-04, 26 May 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-quic-version-negotiation-04>>.
- [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>>.
- [SVCB] Schwartz, B., Bishop, M., and E. Nygren, "Service binding and parameter specification via the DNS (DNS SVCB and HTTPSSVC)", Work in Progress, Internet-Draft, draft-ietf-dnsop-svcb-httpssvc-03, 11 June 2020, <<https://datatracker.ietf.org/doc/html/draft-ietf-dnsop-svcb-httpssvc-03>>.
- [TLS] 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>>.
- [URI] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<https://www.rfc-editor.org/rfc/rfc3986>>.

Appendix A. Acknowledgments

Benjamin Schwartz provided significant input into the design of the mechanism and helped simplify the overall design.

Appendix B. Defining Logical Servers

As incompatible protocols use different protocol stacks, they also use different endpoints. In other words, it is in many cases impossible for the exactly same endpoint to support multiple incompatible protocols. Thus, it is necessary to understand the set of endpoints at a server that offer the incompatible protocols.

A number of choices are possible here:

- * The set of endpoints that are authoritative for the same domain name.
- * The set of endpoints that are authoritative for the same "authority" as defined in RFC 3986 [URI], which is in effect domain name plus port number.
- * The set of endpoints that are referenced by the same SVCB ServiceMode record.
- * The set of endpoints that share an IP address.
- * The set of endpoints that share an IP address and port number.

The challenge with options based on domain name is that it might prevent the use of multiple service providers. This is a common practice for HTTP, where the same domain name can be operated by multiple CDN operators.

Having multiple service operators also rules out using SVCB ServiceMode records also as different records might be used to identify different operators.

Hosts on the same IP address might work, but common deployment practices include use of different ports for entirely different services, which can have different operational constraints such as deployment schedules. Including different ports in the same scope could force all services on the same host to support a consistent set of protocols.

This leaves IP and port. There is always a risk that the same port number is used for completely different purposes depending on the choice of protocol, but this practice is sufficiently rare that it is not anticipated to be a problem.

Author's Address

Martin Thomson
Mozilla

Email: mt@lowentropy.net