

TLS  
Internet-Draft  
Intended status: Experimental  
Expires: May 4, 2020

D. Benjamin  
Google LLC  
November 01, 2019

Batch Signing for TLS  
draft-davidben-tls-batch-signing-02

Abstract

This document describes a mechanism for batch signing in TLS.

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 May 4, 2020.

Copyright Notice

Copyright (c) 2019 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. Conventions and Definitions . . . . .	2
3. Batch SignatureSchemes . . . . .	3
3.1. Signing . . . . .	4
3.2. Verifying . . . . .	6
4. Security Considerations . . . . .	7
4.1. Correctness . . . . .	7
4.2. Domain Separation . . . . .	8
4.3. Payload Confidentiality . . . . .	8
4.4. Information Leaks . . . . .	8
5. IANA Considerations . . . . .	9
6. Normative References . . . . .	9
Appendix A. Test Vectors . . . . .	10
Acknowledgments . . . . .	10
Author's Address . . . . .	10

## 1. Introduction

TLS [RFC8446] clients and servers authenticating with certificates perform online signatures with the private key associated with their certificate. In some cases, signing throughput may be limited. For instance, RSA signing is CPU-intensive compared to many other algorithms used in TLS. The private key may also be stored on a hardware module or be accessed remotely on another server. Under load, this can result in DoS concerns or impact system performance.

To mitigate these concerns, this document introduces a mechanism for batch signing in TLS. It allows TLS implementations to satisfy many concurrent requests with a single signing operation, at a logarithmic cost to signature size. A server under load could, for instance, preferentially serve batch-capable clients as part of its DoS strategy.

## 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 section 3 of [RFC8446].

### 3. Batch SignatureSchemes

A batch SignatureScheme signs a number of input messages from different connections concurrently and returns a corresponding batch signature for each input message.

Each SignatureScheme is parameterized by the following:

- o A base signature algorithm
- o A hash function

This document defines the following values:

```
enum {  
    ecdsa_secp256r1_sha256_batch(TBD1),  
    ecdsa_secp384r1_sha384_batch(TBD2),  
    ecdsa_secp521r1_sha512_batch(TBD3),  
    ed25519_batch(TBD4),  
    ed448_batch(TBD5),  
    rsa_pss_pss_sha256_batch(TBD6),  
    rsa_pss_rsae_sha256_batch(TBD7),  
    rsa_pkcs1_sha256_legacy_batch(TBD8),  
    (65536)  
} SignatureScheme
```

"ecdsa\_secp256r1\_sha256\_batch", "ecdsa\_secp384r1\_sha384\_batch", and "ecdsa\_secp521r1\_sha512\_batch" use base signature algorithms of "ecdsa\_secp256r1\_sha256", "ecdsa\_secp384r1\_sha384", and "ecdsa\_secp521r1\_sha512" with SHA-256, SHA-384, and SHA-512 [SHS], respectively, as the hash function.

"ed25519\_batch" uses a base signature algorithm of "ed25519" with SHA-512 as the hash function. "ed448\_batch" uses a base signature algorithm of "ed448" with 64 bytes (512 bits) of SHAKE256 [FIPS202] output as the hash function.

"rsa\_pss\_pss\_sha256\_batch" and "rsa\_pss\_rsae\_sha256\_batch" use base signature algorithms of "rsa\_pss\_pss\_sha256" and "rsa\_pss\_rsae\_sha256" with SHA-256 as the hash function.

"rsa\_pkcs1\_sha256\_legacy\_batch" uses a base signature algorithm of "rsa\_pkcs1\_sha256\_legacy" [I-D.davidben-tls13-pkcs1] with SHA-256 as the hash function. As with "rsa\_pkcs1\_sha256\_legacy", this code point is only defined for use with client certificates.

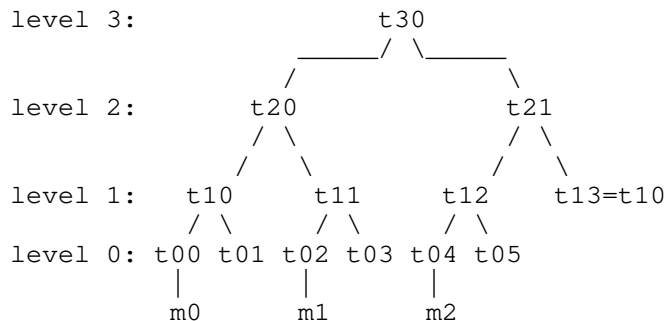
Batch signing is only defined for use with TLS 1.3. If TLS 1.2 is negotiated, the above code points MUST NOT be used in

ServerKeyExchange or CertificateVerify messages. Note, however, a client which supports both TLS 1.2 and TLS 1.3 MAY offer the code points in the ClientHello.

These code points do not correspond to certificate signature algorithms. Implementations wishing to advertise support for the base signature algorithm should send the base algorithm's corresponding code point.

### 3.1. Signing

Signing is performed by building a Merkle tree on top of the signing inputs, interspersed with blinding values. An example tree for three messages is shown below:



In general, let  $n$  be the number of input messages. If  $n$  is greater than  $2^{31}$ , the signing procedure fails and returns an error. Otherwise, it builds a tree with  $l$  levels numbered 0 to  $l-1$ , where  $l$  is  $\text{ceil}(\log_2(n)) + 2$ . Hashes in the tree are built from the following functions:

```

HashLeaf(msg) = Hash(0x00 || msg)
HashNode(left, right) = Hash(0x01 || left || right)
  
```

"0x00" and "0x01" denote byte strings containing a single byte with value zero and one, respectively. "||" denotes concatenation. "left" and "right" are byte strings with length `Hash.length`.

Tree levels are computed iteratively as follows:

1. Initialize level 0 with  $2*n$  elements. For  $i$  between 0 and  $n-1$ , inclusive, set element  $2*i$  to the output of `HashLeaf(m[i])` and element  $2*i+1$  to a random string of `Hash.length` bytes. The random values placed at odd indices preserve signature payload confidentiality (see Section 4.3).

2. For  $i$  between 1 and  $l-1$ , inclusive, compute level  $i$  from level  $i-1$  as follows:
  - \* If level  $i-1$  has an odd number of elements, pad it to an even number of elements with a copy of its first element. That is, if the previous level contained three hashes,  $x, y, z$ , it should now contain four elements,  $x, y, z, x$ .
  - \* Initialize level  $i$  with half as many elements as level  $i-1$ . Set element  $j$  to the output of `HashNode(left, right)` where "left" is element  $2*j$  of level  $i-1$  and "right" is element  $2*j+1$  of level  $i-1$ . "left" and "right" are the left and right children of element  $j$ .

Level  $l-1$  will contain a single element, the root of the tree. The signer then computes a digital signature using the base signature algorithm. This signature is computed over the concatenation of:

- o A string that consists of octet 32 (0x20) repeated 64 times
- o The context string "TLS batch signature"
- o A single 0 byte which serves as the separator
- o The batch signature algorithm's SignatureScheme code point, expressed as a big-endian 16-bit integer. Note this is the code point of the batch algorithm, not the original base algorithm.
- o The value at the root of the tree

This structure is intended to provide key separation with other signatures in TLS (see Section 4.2).

The signer then constructs a BatchSignature structure, as defined below, for each input message. It encodes each to bytes to obtain the final signatures.

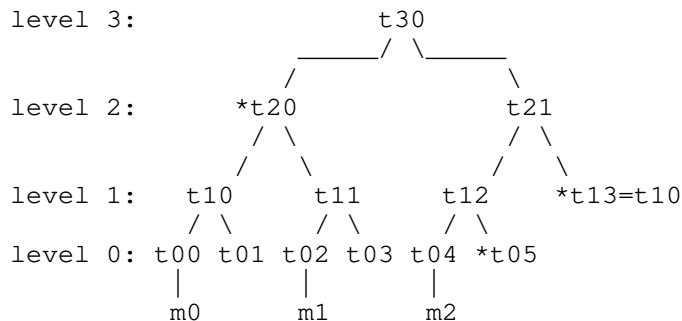
```
opaque Node[Hash.length];

struct {
    uint32 index;
    Node path<Hash.length..2^16-1>;
    opaque root_signature<0..2^16-1>;
} BatchSignature;
```

To assemble the BatchSignature structure for message  $i$ :

1. Set "index" to  $i$ . This will be a value between 0 and  $n-1$ , inclusive.
2. Set "path" to an array of  $l-1$  hashes. Set element  $j$  of this array to element  $k$  of level  $j$ , where  $k$  is  $((2 * i) >> j) ^ 1$ . ">>" denotes a bitwise right-shift, and "^" denotes a bitwise exclusive OR (XOR) operation. This element is the sibling of the ancestor of message  $i$  in the tree. Note the root is never included.
3. Set "root\_signature" to the digital signature computed above.

For example, in the diagram below, the "path" field of the signature of "m2" contains the marked nodes, in order from bottom to top.



### 3.2. Verifying

The signature is verified by recovering the root hash from the supplied "path" and "index" fields and then verifying the signature in the "root\_signature" field. This is done as follows:

1. If decoding the BatchSignature structure fails, terminate the algorithm and reject the signature.
2. If the value of the "index" field is  $2^{31}$  or higher, or if the number of elements in the "path" field is higher than 32, terminate the algorithm and reject the signature. Otherwise, set "remaining" to double this value.
3. Set "hash" to the output of HashLeaf(message).
4. For each element "v" of the "path" field, in order:
  - \* If "remaining" is odd, set "hash" to the output of HashNode(v, hash). Otherwise, set "hash" to the output of HashNode(hash, v)

- \* Set "remaining" to remaining >> 1.
5. If "remaining" is non-zero, the signature is invalid. Terminate the algorithm and reject the signature.
  6. As in the signing algorithm, concatenate the following:
    - \* A string that consists of octet 32 (0x20) repeated 64 times
    - \* The context string "TLS batch signature"
    - \* A single 0 byte which serves as the separator
    - \* The batch signature algorithm's SignatureScheme code point, expressed as a big-endian 16-bit integer. Note this is the code point of the batch algorithm, not the original base algorithm.
    - \* The value of "hash"
  7. Verify that the "root\_signature" field is a valid signature for the concatenation, using the base signature algorithm. If it is invalid, terminate the algorithm and reject the signature. Otherwise, accept the signature.

Note there are many possible valid signatures for a given message, depending on how many and what messages were batched together.

#### 4. Security Considerations

##### 4.1. Correctness

Batch signatures sign the root of a Merkle tree (see Section 3.1) so, provided the hash is collision-resistant and the base algorithm is secure, an attacker can only forge signatures of messages in the leaves of the Merkle tree. These leaves are the input messages, with the exception of padding and blinding nodes, discussed below.

When building the tree, this mechanism pads odd-length levels with extra copies of nodes already in the tree. This is equivalent to signing multiple copies of some input messages to bring the total to a power of two. This avoids introducing other messages for which the signature would also be valid. Verification (see Section 3.2) implicitly rejects odd indices in the tree to likewise ensure blinding values are not mistaken for message hashes.

#### 4.2. Domain Separation

Signatures made by the same key in different contexts should be separated to avoid potential cross-protocol attacks. Inputs to the batch signing algorithm include any existing context strings, such as TLS 1.3's distinct client and server labels or new labels that may be allocated by future versions of TLS. By signing over those labels, batch signing preserves separation between those inputs.

The root signature additionally includes its own context string. This separates it from unbatched TLS 1.3 signatures, defined in section 4.4.3 of [RFC8446]. Like TLS 1.3, it additionally includes a 64-byte padding prefix to clear the ClientHello.random and ServerHello.random prefixes in the TLS 1.2 ServerKeyExchange signing payload. This allows the same key to be used for batched and unbatched signatures, simplifying deployment.

Finally, including the code point in the signature payload provides separation in case the same base signature algorithm is used in two batch constructions with, say, different hash functions.

#### 4.3. Payload Confidentiality

The signing payload in TLS 1.3 is the handshake transcript. This contains information which is normally encrypted, such as the server certificate. Path elements in a batch signature are computed from payloads from other connections in the same batch. A naive construction could permit one peer to learn confidential information in other connections' signing payloads, such as which server certificate was selected in response to an encrypted SNI.

This mechanism avoids these attacks by pairing each input with a secret blinding value. An input's signature path will reveal the corresponding blinding value at level 0, but all other inputs in the path are incorporated in nodes at level 1 or higher. Provided the hash is preimage-resistant, these nodes do not reveal the original payload.

In the event of entropy failure when generating the blinding values, signatures remain unforgeable. The blinding values are only needed for payload confidentiality.

#### 4.4. Information Leaks

A server observing multiple batched client signatures with the same root hash learns the two connections were created by the same client. However, the connections are already correlatable via the client certificate itself, so this does not reveal additional information in



most deployments. Clients can partition the contexts in which signing requests may be batched to further mitigate these issues.

Additionally, a single batch signature reveals the number of signing requests in that batch, rounded up to a power of two. This may reveal some information about a service's signing load.

## 5. IANA Considerations

IANA is requested to create the following entries in the TLS SignatureScheme registry, defined in [RFC8446]. The "Reference" column should be set to this document.

Value	Description	Recommended
TBD1	ecdsa_secp256r1_sha256_batch	Y
TBD2	ecdsa_secp384r1_sha384_batch	Y
TBD3	ecdsa_secp521r1_sha512_batch	Y
TBD4	ed25519_batch	Y
TBD5	ed448_batch	Y
TBD6	rsa_pss_pss_sha256_batch	Y
TBD7	rsa_pss_rsae_sha256_batch	Y
TBD8	rsa_pkcs1_sha256_legacy_batch	N

## 6. Normative References

[FIPS202] Dworkin, M., "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.202, July 2015.

[I-D.davidben-tls13-pkcs1]  
Benjamin, D., "Legacy RSASSA-PKCS1-v1\_5 codepoints for TLS 1.3", draft-davidben-tls13-pkcs1-00 (work in progress), July 2019.

- [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>>.
- [SHS] Dang, Q., "Secure Hash Standard", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.180-4, July 2015.

#### Appendix A. Test Vectors

TODO: Include test vectors. Probably use `ecdsa_secp256r1_sha256_batch`. RSA signatures are big and Ed25519 isn't as common. Include some negative examples for verifying as well as intermediate values so signing code can at least compare against the tree-building vectors. (Blinding values and most of our defined signature schemes are non-deterministic.)

#### Acknowledgments

The mechanism described in this document is derived from a similar construction by Adam Langley in the Roughtime protocol. Adam also provided the initial suggestion to apply a similar technique to TLS.

#### Author's Address

David Benjamin  
Google LLC

Email: [davidben@google.com](mailto:davidben@google.com)

TLS  
Internet-Draft  
Intended status: Experimental  
Expires: 3 June 2022

S. Farrell  
Trinity College Dublin  
30 November 2021

A well-known URI for publishing ECHConfigList values.  
draft-farrell-tls-wkesni-02

## Abstract

We propose use of a well-known URI at which web servers can publish ECHConfigList values as a way to help get those published in the DNS.

## 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 3 June 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 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. Terminology . . . . .	3
3. Example use of the well-known URI for ECH . . . . .	3
4. The ech well-known URI . . . . .	4
5. The JSON structure for ECHConfigList values . . . . .	4
6. Zone factory behaviour . . . . .	5
7. Security Considerations . . . . .	6
8. Acknowledgements . . . . .	6
9. IANA Considerations . . . . .	6
10. Normative References . . . . .	6
Appendix A. Change Log . . . . .	7
Author's Address . . . . .	7

## 1. Introduction

Encrypted ClientHello (ECH) [I-D.ietf-tls-esni] for TLS1.3 [RFC8446] defines a confidentiality mechanism for server names and other ClientHello content in TLS. For many applications, that requires publication of ECHConfigList data structures in the DNS. An ECHConfigList structure contains a list of ECHConfig values. Each ECHConfig value contains the public component of a key pair that will typically be periodically (re-)generated by a web server. Many web infrastructures will have an API that can be used to dynamically update the DNS RR values containing ECHConfigList values. Some deployments however, will not, so web deployments could benefit from a mechanism to use in such cases.

We define such a mechanism here. Note that this is not intended for universal deployment, but rather for cases where the web server doesn't have write access to the relevant zone file (or equivalent). That zone file will eventually include an HTTPS or SVCB RR [I-D.ietf-dnsop-svcb-https] containing an ECHConfigList.

We use the term "zone factory" for the entity that does have write access to the zone file. We assume the zone factory (ZF) can also make HTTPS requests to the web server with the ECH keys.

We propose use of a well-known URI [RFC8615] on the web server that allows ZF to poll for changes to ECHConfigList values. For example, if a web server generates new ECHConfigList values hourly and publishes those at the well-known URI, ZF can poll that URI. When ZF sees new values, it can check if those work, and if they do, then update the zone file and re-publish the zone.

[[This idea could: a) wither on the vine, b) be published as it's own RFC, or c) end up as a PR for [I-D.ietf-tls-esni]. There is no absolute need for this to be in the RFC that defines ECH, so (b) seems feasible if there's enough interest, hence this draft. The source for this is in <https://github.com/sftcd/wkesni/> PRs are welcome there too.]]

## 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.

## 3. Example use of the well-known URI for ECH

An example deployment could be as follows:

- \* Web server generates new ECHConfigList values hourly at N past the hour via a cronjob
- \* ECHConfigList values are "current" for an hour, published with a TTL of 1800, and remain usable for 3 hours from the time of generation
- \* Web server has a set of "backend" sites - the DNS name for each such site is here represented as \$BACKEND, which will end up as an SNI value to be encrypted inside an ECH extension
- \* Web server has a "front-end" site (\$FRONT), where \$FRONT will typically be the DNS name used in the ECHConfigList public\_name field for ECHConfig version 0xff0d
- \* A cronjob creates creates a JSON file for each backend site at [https://\\$FRONT/.well-known/ech/\\$BACKEND.json](https://$FRONT/.well-known/ech/$BACKEND.json)
- \* Each JSON file contains an array with the ECHConfigList values values for that particular \$BACKEND as shown in Figure 1 - the values in Figure 1 with ellipses are the values we want to eventually see in the DNS
- \* On the zone factory, a cronjob runs at N+3 past the hour, it knows all the names involved and checks to see if the content at those well-known URIs has changed or not
- \* If the content has changed the cronjob attempts to use the ECHConfigList values, and for each \$BACKEND where that works, it updates the zone file and re-publishes the zone containing only the new ECHConfigList values

#### 4. The ech well-known URI

When a web server (\$FRONT) wants to publish ECHConfigList information for a backend site (\$BACKEND) then it provides the JSON content defined in Section 5 at: `https://$FRONT/.well-known/ech/$BACKEND.json`

The well-known URI defined here MUST be an https URL and therefore the zone factory verifies the correct \$FRONT is being accessed. If there is any failure in accessing the well-known URI, then the zone factory MUST NOT modify the zone.

#### 5. The JSON structure for ECHConfigList values

[[Since the specifics of the JSON structure in Figure 1 are very likely to change, this is mostly TBD. What is here for now, is what the author has currently implemented simply because it worked ok and was easy to do:-)]]

[[Might change this due to retry-configs and now that I've implemented split-mode]]

```
[
  {
    "desired-ttl": 1800,
    "ports": [ 443, 8413 ],
    "echconfiglist": "AD7+DQA65wAgAC..AA=="
  },
  {
    "desired-ttl": 1800,
    "ports": [ 443, 8413 ],
    "echconfiglist": "AD7+DQA65wAgAC..AA=="
  }
]
```

Figure 1: Sample JSON

The JSON file at the well-known URI MUST contain an array with one or more elements. Each element of the array MUST have these fields:

- \* `desired-ttl`: contains a number indicating the TTL that the web server would like to see used for this RR. The zone factory MUST NOT use a longer TTL.
- \* `ports`: this has a list of the TCP ports on which the web server with the relevant key pair will listen (needed to produce the correct zone file).
- \* `ECHConfigList`: contains the value to be used as a base64 encoded string.

The JSON file contains an array for a couple of reasons:

- \* As TLS authentication doesn't really distinguish ports, servers on the same host could in any case cheat on one another, so we may as well just read one JSON file per name.
- \* Different ports could map to different sets of ECHConfigList values
- \* As ECHConfigList is (regrettably:-) an extensible structure, it may be necessary to publish different ECHConfigList values to get best interoperability.

## 6. Zone factory behaviour

The zone factory SHOULD check that the presented ECHConfigList values work with the \$BACKEND server before publication. A "special" TLS client may be needed for this check, that does not require the ECHConfigList value to have already been published in the DNS. [[I guess that calls for the zone factory to know of a "safe" URL on \$BACKEND to try, or maybe it could use HTTP HEAD? Figuring that out is TBD. The ZF could also try a GREASEd ECH and see if the retry-configs it gets back is one of the ECHConfigList values in the ECHConfigList.]]

A careful zone factory could explode the ECHConfigList value presented into "singleton" values with one public key in each and test each for each port claimed.

The zone factory SHOULD publish all the ECHConfigList values that are presented in the JSON file, and that pass the check above.

The zone factory SHOULD only publish ECHConfigList values that are in the latest version of the JSON file. This leaves the control of "expiry" with the web server, so long as the ECHConfigList values presented actually work. [[An alternative could be to have the new values just be appended to the zone, but that'd require some form of "notAfter" value in the JSON file which seems unnecessary and more complex.]]

The SCVB and HTTPS RR specification [I-D.ietf-dnsop-svcb-https] defines how and where the ECHConfigList values for \$BACKEND needs to be published in the DNS. The zone factory is assumed to be in control of how ECHConfigList values are included in such RRs.

A possibly interesting (unintended) consequence of this design is that once a TLS client has first gotten an ECHConfigList from the DNS for \$BACKEND with the ECHConfigList structure containing the public\_name field, the TLS client would know both \$FRONT and \$BACKEND and so could later probe for this .well-known as an alternative to doing so via DoT/DoH. Probably not something a web browser might do, but could be fun for other applications maybe.

[[The extent to which retry-configs could be used for a similar purpose might be worth considering. But the JSON stuff here may still be needed if implementations (such as mine:-) tend to only return one ECHConfig in retry-configs.]]

## 7. Security Considerations

This document defines another way to publish ECHConfigList values. If the wrong keys were read from here and published in the DNS, then clients using ECH would do the wrong thing, likely resulting in denial of service, or a privacy leak, or worse, when TLS clients attempt to use ECH with a backend web site. So: Don't do that:-)

## 8. Acknowledgements

Thanks to Niall O'Reilly for a quick review of -00.

## 9. IANA Considerations

[[TBD: IANA registration of a .well-known. Also TBD - how to handle I18N for \$FRONT and \$BACKEND within such a URL.]]

## 10. 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>>.
- [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>>.
- [RFC8615] Nottingham, M., "Well-Known Uniform Resource Identifiers (URIs)", RFC 8615, DOI 10.17487/RFC8615, May 2019, <<https://www.rfc-editor.org/info/rfc8615>>.



[I-D.ietf-tls-esni]

Rescorla, E., Oku, K., Sullivan, N., and C. A. Wood, "TLS Encrypted Client Hello", Work in Progress, Internet-Draft, draft-ietf-tls-esni-13, 12 August 2021, <<https://www.ietf.org/archive/id/draft-ietf-tls-esni-13.txt>>.

[I-D.ietf-dnsop-svcb-https]

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>>.

#### Appendix A. Change Log

[[RFC editor: please remove this before publication.]]

From -01 to -02:

- \* General changes from ESNI to ECH.

From -00 to -01:

- \* Re-structured a bit after re-reading rfc8615

#### Author's Address

Stephen Farrell  
Trinity College Dublin  
Dublin  
2  
Ireland

Phone: +353-1-896-2354  
Email: [stephen.farrell@cs.tcd.ie](mailto:stephen.farrell@cs.tcd.ie)

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

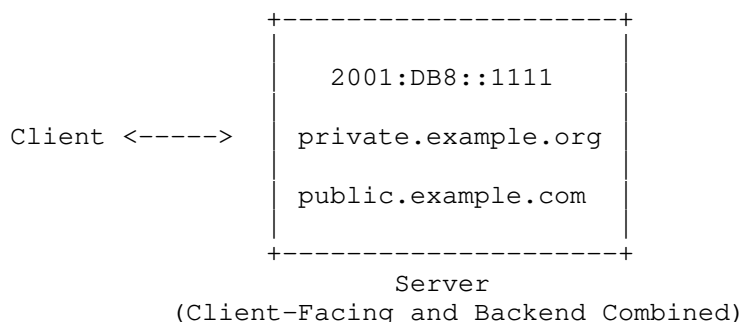


Figure 1: Shared Mode Topology

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.

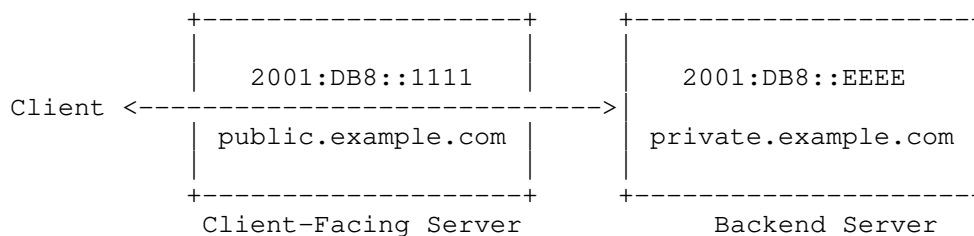


Figure 2: Split Mode Topology

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.

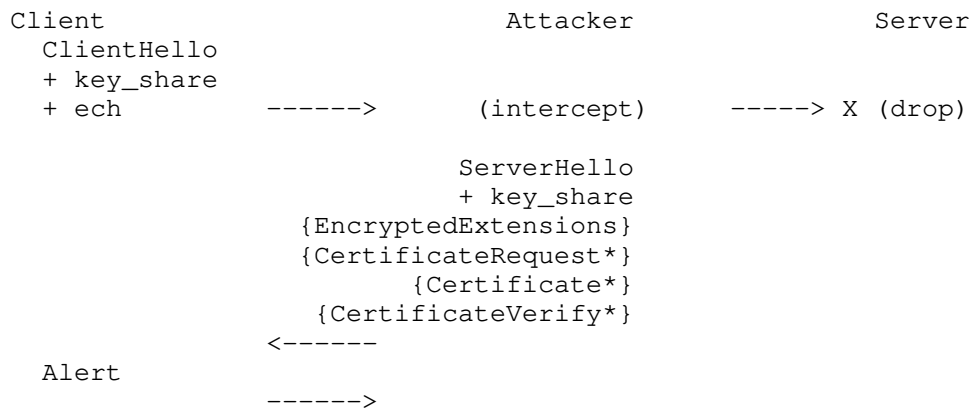


Figure 3: Client reaction attack

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.

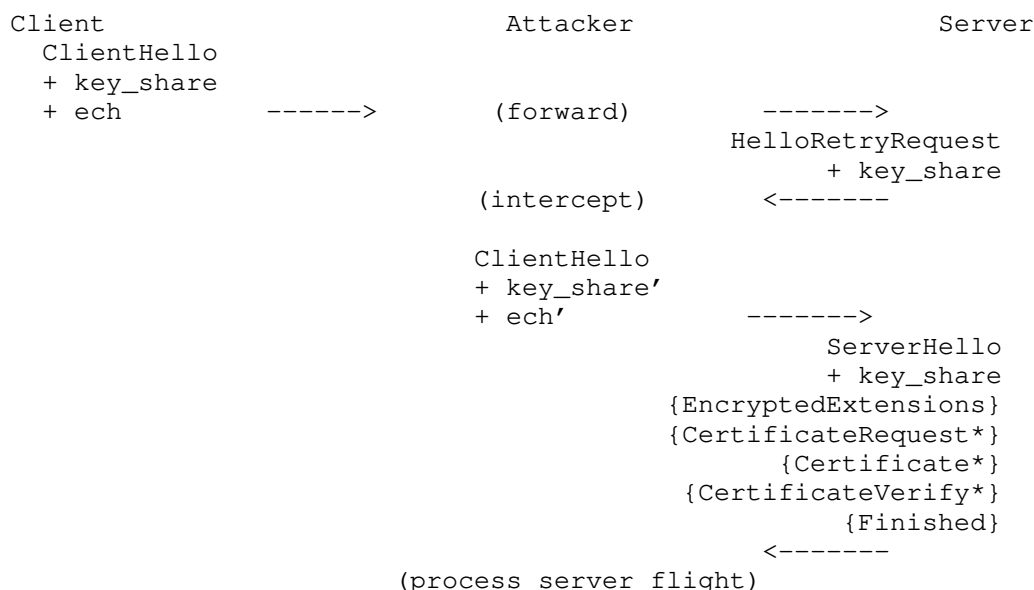


Figure 4: HelloRetryRequest hijack attack

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.



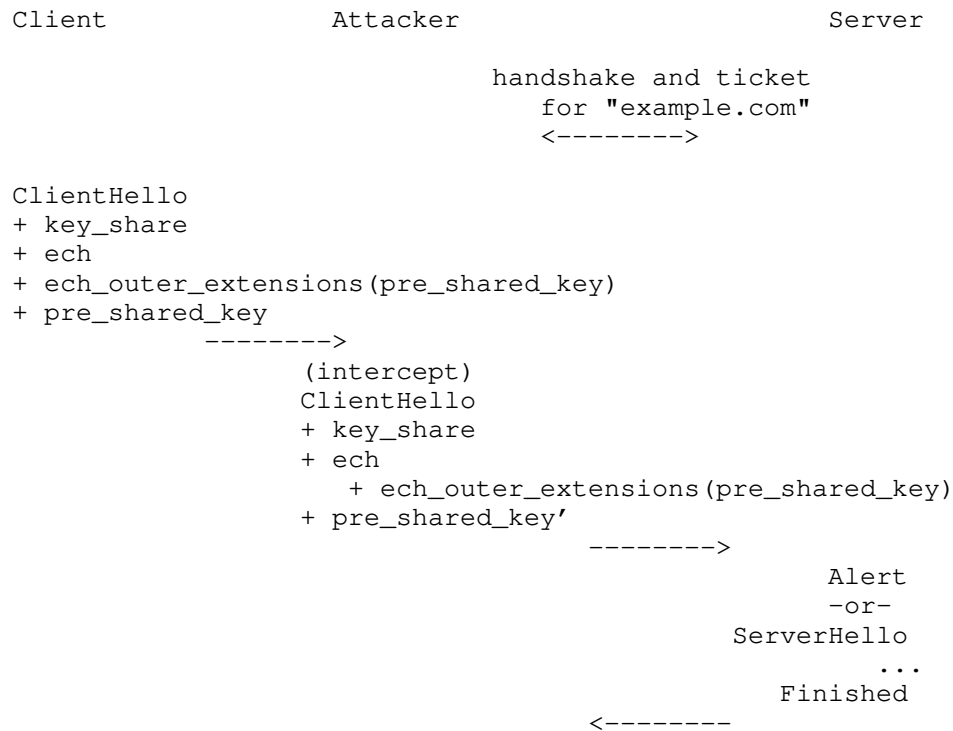


Figure 5: Message flow for malleable ClientHello

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](mailto:ekr@rtfm.com)

Kazuho Oku  
Fastly

Email: [kazuhooku@gmail.com](mailto:kazuhooku@gmail.com)



Nick Sullivan  
Cloudflare

Email: [nick@cloudflare.com](mailto:nick@cloudflare.com)

Christopher A. Wood  
Cloudflare

Email: [caw@heapingbits.net](mailto:caw@heapingbits.net)

TLS  
Internet-Draft  
Intended status: Standards Track  
Expires: 5 September 2022

N. Sullivan  
Cloudflare Inc.  
4 March 2022

Exported Authenticators in TLS  
draft-ietf-tls-exported-authenticator-15

Abstract

This document describes a mechanism that builds on Transport Layer Security (TLS) or Datagram Transport Layer Security (DTLS) and enables peers to provide a proof of ownership of an identity, such as an X.509 certificate. This proof can be exported by one peer, transmitted out-of-band to the other peer, and verified by the receiving peer.

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 5 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 Terminology . . . . .	3
3. Message Sequences . . . . .	4
4. Authenticator Request . . . . .	4
5. Authenticator . . . . .	6
5.1. Authenticator Keys . . . . .	6
5.2. Authenticator Construction . . . . .	7
5.2.1. Certificate . . . . .	8
5.2.2. CertificateVerify . . . . .	8
5.2.3. Finished . . . . .	10
5.2.4. Authenticator Creation . . . . .	10
6. Empty Authenticator . . . . .	10
7. API considerations . . . . .	11
7.1. The "request" API . . . . .	11
7.2. The "get context" API . . . . .	11
7.3. The "authenticate" API . . . . .	11
7.4. The "validate" API . . . . .	12
8. IANA Considerations . . . . .	13
8.1. Update of the TLS ExtensionType Registry . . . . .	13
8.2. Update of the TLS Exporter Labels Registry . . . . .	13
8.3. Update of the TLS HandshakeType Registry . . . . .	13
9. Security Considerations . . . . .	13
10. Acknowledgements . . . . .	14
11. References . . . . .	14
11.1. Normative References . . . . .	14
11.2. Informative References . . . . .	15
Author's Address . . . . .	16

## 1. Introduction

This document provides a way to authenticate one party of a Transport Layer Security (TLS) or Datagram Transport Layer Security (DTLS) connection to its peer using authentication messages created after the session has been established. This allows both the client and server to prove ownership of additional identities at any time after the handshake has completed. This proof of authentication can be exported and transmitted out-of-band from one party to be validated by its peer.

This mechanism provides two advantages over the authentication that TLS and DTLS natively provide:

multiple identities - Endpoints that are authoritative for multiple identities - but do not have a single certificate that includes all of the identities - can authenticate additional identities over a single connection.

spontaneous authentication - Endpoints can authenticate after a connection is established, in response to events in a higher-layer protocol, as well as integrating more context (such as context from the application).

Versions of TLS prior to TLS 1.3 used renegotiation as a way to enable post-handshake client authentication given an existing TLS connection. The mechanism described in this document may be used to replace the post-handshake authentication functionality provided by renegotiation. Unlike renegotiation, exported Authenticator-based post-handshake authentication does not require any changes at the TLS layer.

Post-handshake authentication is defined in section 4.6.3 of TLS 1.3 [RFC8446], but it has the disadvantage of requiring additional state to be stored as part of the TLS state machine. Furthermore, the authentication boundaries of TLS 1.3 post-handshake authentication align with TLS record boundaries, which are often not aligned with the authentication boundaries of the higher-layer protocol. For example, multiplexed connection protocols like HTTP/2 [RFC7540] do not have a notion of which TLS record a given message is a part of.

Exported Authenticators are meant to be used as a building block for application protocols. Mechanisms such as those required to advertise support and handle authentication errors are not handled by TLS (or DTLS).

The minimum version of TLS and DTLS required to implement the mechanisms described in this document are TLS 1.2 [RFC6347] and DTLS 1.2 [RFC5246].

## 2. Conventions and Terminology

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

This document uses terminology such as client, server, connection, handshake, endpoint, peer that are defined in section 1.1 of [RFC8446]. The term "initial connection" refers to the (D)TLS connection from which the exported authenticator messages are derived.

### 3. Message Sequences

There are two types of messages defined in this document: Authenticator Requests and Authenticators. These can be combined in the following three sequences:

#### Client Authentication

- \* Server generates Authenticator Request
- \* Client generates Authenticator from Server's Authenticator Request
- \* Server validates Client's Authenticator

#### Server Authentication

- \* Client generates Authenticator Request
- \* Server generates Authenticator from Client's Authenticator Request
- \* Client validates Server's Authenticator

#### Spontaneous Server Authentication

- \* Server generates Authenticator
- \* Client validates Server's Authenticator

### 4. Authenticator Request

The authenticator request is a structured message that can be created by either party of a (D)TLS connection using data exported from that connection. It can be transmitted to the other party of the (D)TLS connection at the application layer. The application layer protocol used to send the authenticator request SHOULD use a secure transport channel with equivalent security to TLS, such as QUIC [RFC9001], as its underlying transport to keep the request confidential. The application MAY use the existing (D)TLS connection to transport the authenticator.

An authenticator request message can be constructed by either the client or the server. Server-generated authenticator requests use the `CertificateRequest` message from Section 4.3.2 of [RFC8446]. Client-generated authenticator requests use a new message, called the `ClientCertificateRequest`, which uses the same structure as `CertificateRequest`. (Note that the latter is not a request for a client certificate, but rather a certificate request generated by the client.) These message structures are used even if the connection protocol is TLS 1.2 or DTLS 1.2.

The `CertificateRequest` and `ClientCertificateRequest` messages are used to define the parameters in a request for an authenticator. These are encoded as TLS handshake messages, including length and type fields. They do not include any TLS record layer framing and are not encrypted with a handshake or application-data key.

The structures are defined to be:

```
struct {
    opaque certificate_request_context<0..2^8-1>;
    Extension extensions<2..2^16-1>;
} ClientCertificateRequest;

struct {
    opaque certificate_request_context<0..2^8-1>;
    Extension extensions<2..2^16-1>;
} CertificateRequest;
```

`certificate_request_context`: An opaque string which identifies the authenticator request and which will be echoed in the authenticator message. A `certificate_request_context` value MUST be unique for each authenticator request within the scope of a connection (preventing replay and context confusion). The `certificate_request_context` SHOULD be chosen to be unpredictable to the peer (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the peer's private key from pre-computing valid authenticators. For example, the application may choose this value to correspond to a value used in an existing datastructure in the software to simplify implementation.

`extensions`: The set of extensions allowed in the `CertificateRequest` structure and the `ClientCertificateRequest` structure are those defined in the TLS ExtensionType Values IANA registry [RFC8447] containing CR in the TLS 1.3 column. In addition, the set of extensions in the `ClientCertificateRequest` structure MAY include the `server_name` [RFC6066] extension.

The uniqueness requirements of the `certificate_request_context` apply only to `CertificateRequest` and `ClientCertificateRequest` messages that are used as part of authenticator requests, but do apply across `CertificateRequest` and `ClientCertificateRequest` messages. A `certificate_request_context` value used in a `ClientCertificateRequest` cannot be used in an authenticator `CertificateRequest` on the same connection, and vice versa. There is no impact if the value of a `certificate_request_context` used in an authenticator request matches the value of a `certificate_request_context` in the handshake or in a post-handshake message.

## 5. Authenticator

The authenticator is a structured message that can be exported from either party of a (D)TLS connection. It can be transmitted to the other party of the (D)TLS connection at the application layer. The application layer protocol used to send the authenticator SHOULD use a secure transport channel with equivalent security to TLS, such as QUIC [RFC9001], as its underlying transport to keep the authenticator confidential. The application MAY use the existing (D)TLS connection to transport the authenticator.

An authenticator message can be constructed by either the client or the server given an established (D)TLS connection, an identity, such as an X.509 certificate, and a corresponding private key. Clients MUST NOT send an authenticator without a preceding authenticator request; for servers an authenticator request is optional. For authenticators that do not correspond to authenticator requests, the `certificate_request_context` is chosen by the server.

### 5.1. Authenticator Keys

Each authenticator is computed using a Handshake Context and Finished MAC Key derived from the (D)TLS connection. These values are derived using an exporter as described in Section 4 of [RFC5705] (for (D)TLS 1.2) or Section 7.5 of [RFC8446] (for (D)TLS 1.3). For (D)TLS 1.3, the `exporter_master_secret` MUST be used, not the `early_exporter_master_secret`. These values use different labels depending on the role of the sender:

- \* The Handshake Context is an exporter value that is derived using the label "EXPORTER-client authenticator handshake context" or "EXPORTER-server authenticator handshake context" for authenticators sent by the client or server respectively.

- \* The Finished MAC Key is an exporter value derived using the label "EXPORTER-client authenticator finished key" or "EXPORTER-server authenticator finished key" for authenticators sent by the client or server respectively.

The context\_value used for the exporter is empty (zero length) for all four values. There is no need to include additional context information at this stage since the application-supplied context is included in the authenticator itself. The length of the exported value is equal to the length of the output of the hash function associated with the selected cipher suite (for TLS 1.3) or the hash function used for the pseudorandom function (PRF) (for (D)TLS 1.2). Exported authenticators cannot be used with (D)TLS 1.2 cipher suites that do not use the TLS PRF and with TLS 1.3 cipher suites that do not have an associated hash function. This hash is referred to as the authenticator hash.

To avoid key synchronization attacks, Exported Authenticators MUST NOT be generated or accepted on (D)TLS 1.2 connections that did not negotiate the extended master secret extension [RFC7627].

## 5.2. Authenticator Construction

An authenticator is formed from the concatenation of TLS 1.3 [RFC8446] Certificate, CertificateVerify, and Finished messages. These messages are encoded as TLS handshake messages, including length and type fields. They do not include any TLS record layer framing and are not encrypted with a handshake or application-data key.

If the peer populating the certificate\_request\_context field in an authenticator's Certificate message has already created or correctly validated an authenticator with the same value, then no authenticator should be constructed. If there is no authenticator request, the extensions are chosen from those presented in the (D)TLS handshake's ClientHello. Only servers can provide an authenticator without a corresponding request.

ClientHello extensions are used to determine permissible extensions in the server's unsolicited Certificate message in order to follow the general model for extensions in (D)TLS in which extensions can only be included as part of a Certificate message if they were previously sent as part of a CertificateRequest message or ClientHello message. This ensures that the recipient will be able to process such extensions.



### 5.2.1. Certificate

The Certificate message contains the identity to be used for authentication, such as the end-entity certificate and any supporting certificates in the chain. This structure is defined in [RFC8446], Section 4.4.2.

The Certificate message contains an opaque string called `certificate_request_context`, which is extracted from the authenticator request if present. If no authenticator request is provided, the `certificate_request_context` can be chosen arbitrarily but MUST be unique within the scope of the connection and be unpredictable to the peer.

Certificates chosen in the Certificate message MUST conform to the requirements of a Certificate message in the negotiated version of (D)TLS. In particular, the entries of `certificate_list` MUST be valid for the signature algorithms indicated by the peer in the `"signature_algorithms"` and `"signature_algorithms_cert"` extension, as described in Section 4.2.3 of [RFC8446] for (D)TLS 1.3 or from Sections 7.4.2 and 7.4.6 of [RFC5246] for (D)TLS 1.2.

In addition to `"signature_algorithms"` and `"signature_algorithms_cert"`, the `"server_name"` [RFC6066], `"certificate_authorities"` (Section 4.2.4. of [RFC8446]), and `"oid_filters"` (Section 4.2.5. of [RFC8446]) extensions are used to guide certificate selection.

Only the X.509 certificate type defined in [RFC8446] is supported. Alternative certificate formats such as [RFC7250] Raw Public Keys are not supported in this version of the specification and their use in this context has not yet been analysed.

If an authenticator request was provided, the Certificate message MUST contain only extensions present in the authenticator request. Otherwise, the Certificate message MUST contain only extensions present in the (D)TLS ClientHello. Unrecognized extensions in the authenticator request MUST be ignored.

### 5.2.2. CertificateVerify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its identity. The format of this message is taken from TLS 1.3:

```
struct {  
    SignatureScheme algorithm;  
    opaque signature<0..2^16-1>;  
} CertificateVerify;
```

The algorithm field specifies the signature algorithm used (see Section 4.2.3 of [RFC8446] for the definition of this field). The signature is a digital signature using that algorithm.

The signature scheme MUST be a valid signature scheme for TLS 1.3. This excludes all RSASSA-PKCS1-v1\_5 algorithms and combinations of ECDSA and hash algorithms that are not supported in TLS 1.3.

If an authenticator request is present, the signature algorithm MUST be chosen from one of the signature schemes present in the "signature\_algorithms" extension of the authenticator request. Otherwise, with spontaneous server authentication, the signature algorithm used MUST be chosen from the "signature\_algorithms" sent by the peer in the ClientHello of the (D)TLS handshake. If there are no available signature algorithms, then no authenticator should be constructed.

The signature is computed using the chosen signature scheme over the concatenation of:

- \* A string that consists of octet 32 (0x20) repeated 64 times
- \* The context string "Exported Authenticator" (which is not NUL-terminated)
- \* A single 0 octet which serves as the separator
- \* The hashed authenticator transcript

The authenticator transcript is the hash of the concatenated Handshake Context, authenticator request (if present), and Certificate message:

Hash(Handshake Context || authenticator request || Certificate)

Where Hash is the authenticator hash defined in section 4.1. If the authenticator request is not present, it is omitted from this construction, i.e., it is zero-length.

If the party that generates the exported authenticator does so with a different connection than the party that is validating it, then the Handshake Context will not match, resulting in a CertificateVerify message that does not validate. This includes situations in which

the application data is sent via TLS-terminating proxy. Given a failed CertificateVerify validation, it may be helpful for the application to confirm that both peers share the same connection using a value derived from the connection secrets (such as the Handshake Context) before taking a user-visible action.

#### 5.2.3. Finished

An HMAC [HMAC] over the hashed authenticator transcript, which is the concatenation of the Handshake Context, authenticator request (if present), Certificate, and CertificateVerify. The HMAC is computed using the authenticator hash, using the Finished MAC Key as a key.

```
Finished = HMAC(Finished MAC Key, Hash(Handshake Context ||  
    authenticator request || Certificate || CertificateVerify))
```

#### 5.2.4. Authenticator Creation

An endpoint constructs an authenticator by serializing the Certificate, CertificateVerify, and Finished as TLS handshake messages and concatenating the octets:

```
Certificate || CertificateVerify || Finished
```

An authenticator is valid if the CertificateVerify message is correctly constructed given the authenticator request (if used) and the Finished message matches the expected value. When validating an authenticator, constant-time comparisons SHOULD be used for signature and MAC validation.

### 6. Empty Authenticator

If, given an authenticator request, the endpoint does not have an appropriate identity or does not want to return one, it constructs an authenticated refusal called an empty authenticator. This is a Finished message sent without a Certificate or CertificateVerify. This message is an HMAC over the hashed authenticator transcript with a Certificate message containing no CertificateEntries and the CertificateVerify message omitted. The HMAC is computed using the authenticator hash, using the Finished MAC Key as a key. This message is encoded as a TLS handshake message, including length and type field. It does not include TLS record layer framing and is not encrypted with a handshake or application-data key.

```
Finished = HMAC(Finished MAC Key, Hash(Handshake Context ||  
    authenticator request || Certificate))
```

## 7. API considerations

The creation and validation of both authenticator requests and authenticators SHOULD be implemented inside the (D)TLS library even if it is possible to implement it at the application layer. (D)TLS implementations supporting the use of exported authenticators SHOULD provide application programming interfaces by which clients and servers may request and verify exported authenticator messages.

Notwithstanding the success conditions described below, all APIs MUST fail if:

- \* the connection uses a (D)TLS version of 1.1 or earlier, or
- \* the connection is (D)TLS 1.2 and the extended master secret extension [RFC7627] was not negotiated

The following sections describe APIs that are considered necessary to implement exported authenticators. These are informative only.

### 7.1. The "request" API

The "request" API takes as input:

- \* `certificate_request_context` (from 0 to 255 octets)
- \* set of extensions to include (this MUST include `signature_algorithms`) and the contents thereof

It returns an authenticator request, which is a sequence of octets that comprises a `CertificateRequest` or `ClientCertificateRequest` message.

### 7.2. The "get context" API

The "get context" API takes as input:

- \* authenticator or authenticator request

It returns the `certificate_request_context`.

### 7.3. The "authenticate" API

The "authenticate" API takes as input:

- \* a reference to the initial connection

- \* an identity, such as a set of certificate chains and associated extensions (OCSP [RFC6960], SCT [RFC6962], etc.)
- \* a signer (either the private key associated with the identity, or interface to perform private key operations) for each chain
- \* an authenticator request or `certificate_request_context` (from 0 to 255 octets)

It returns either the exported authenticator or an empty authenticator as a sequence of octets. It is recommended that the logic for selecting the certificates and extensions to include in the exporter is implemented in the TLS library. Implementing this in the TLS library lets the implementer take advantage of existing extension and certificate selection logic and more easily remember which extensions were sent in the ClientHello.

It is also possible to implement this API outside of the TLS library using TLS exporters. This may be preferable in cases where the application does not have access to a TLS library with these APIs or when TLS is handled independently of the application layer protocol.

#### 7.4. The "validate" API

The "validate" API takes as input:

- \* a reference to the initial connection
- \* an optional authenticator request
- \* an authenticator
- \* a function for validating a certificate chain

It returns a status to indicate whether the authenticator is valid or not after applying the function for validating the certificate chain to the chain contained in the authenticator. If validation is successful, it also returns the identity, such as the certificate chain and its extensions.

The API should return a failure if the `certificate_request_context` of the authenticator was used in a different authenticator that was previously validated. Well-formed empty authenticators are returned as invalid.

When validating an authenticator, constant-time comparison should be used.

## 8. IANA Considerations

### 8.1. Update of the TLS ExtensionType Registry

IANA is requested to update the entry for `server_name(0)` in the registry for ExtensionType (defined in [RFC8446]) by replacing the value in the "TLS 1.3" column with the value "CH, EE, CR" and adding this document in the "Reference" column.

IANA is also requested to add the following note to the registry:

The addition of the "CR" to the "TLS 1.3" column for the `server_name(0)` extension only marks the extension as valid in a ClientCertificateRequest created as part of client-generated authenticator requests.

### 8.2. Update of the TLS Exporter Labels Registry

IANA is requested to add the following entries to the registry for Exporter Labels (defined in [RFC5705]): "EXPORTER-client authenticator handshake context", "EXPORTER-server authenticator handshake context", "EXPORTER-client authenticator handshake context", "EXPORTER-client authenticator finished key" and "EXPORTER-server authenticator finished key" with "DTLS-OK" and "Recommended" set to "Y" and this document added to the "Reference" column.

### 8.3. Update of the TLS HandshakeType Registry

IANA is requested to add the following entry to the registry for HandshakeType (defined in [RFC8446]): "client\_certificate\_request" with "DTLS-OK" and "Recommended" set to "Y" and this document added to the "Reference" column with the following in the "Note" column: "Used in TLS versions prior to 1.3."

## 9. Security Considerations

The Certificate/Verify/Finished pattern intentionally looks like the TLS 1.3 pattern which now has been analyzed several times. For example, [SIGMAC] presents a relevant framework for analysis, and section 10. of [RFC8446] contains a comprehensive set of references.

Authenticators are independent and unidirectional. There is no explicit state change inside TLS when an authenticator is either created or validated. The application in possession of a validated authenticator can rely on any semantics associated with data in the `certificate_request_context`.

- \* This property makes it difficult to formally prove that a server is jointly authoritative over multiple identities, rather than individually authoritative over each.
- \* There is no indication in (D)TLS about which point in time an authenticator was computed. Any feedback about the time of creation or validation of the authenticator should be tracked as part of the application layer semantics if required.

The signatures generated with this API cover the context string "Exported Authenticator" and therefore cannot be transplanted into other protocols.

In TLS 1.3 the client can not explicitly learn from the TLS layer whether its Finished message was accepted. Because the application traffic keys are not dependent on the client's final flight, receiving messages from the server does not prove that the server received the client's Finished. To avoid disagreement between the client and server on the authentication status of EAs, servers MUST verify the client Finished before sending an EA or processing a received EA.

## 10. Acknowledgements

Comments on this proposal were provided by Martin Thomson.  
Suggestions for Section 9 were provided by Karthikeyan Bhargavan.

## 11. References

### 11.1. Normative References

- [HMAC] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <<https://www.rfc-editor.org/info/rfc2104>>.
- [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>>.
- [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>>.

- [RFC5705] Rescorla, E., "Keying Material Exporters for Transport Layer Security (TLS)", RFC 5705, DOI 10.17487/RFC5705, March 2010, <<https://www.rfc-editor.org/info/rfc5705>>.
- [RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <<https://www.rfc-editor.org/info/rfc6066>>.
- [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>>.
- [RFC7627] Bhargavan, K., Ed., Delignat-Lavaud, A., Pironti, A., Langley, A., and M. Ray, "Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension", RFC 7627, DOI 10.17487/RFC7627, September 2015, <<https://www.rfc-editor.org/info/rfc7627>>.
- [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>>.
- [RFC8447] Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", RFC 8447, DOI 10.17487/RFC8447, August 2018, <<https://www.rfc-editor.org/info/rfc8447>>.

## 11.2. Informative References

- [RFC6960] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 6960, DOI 10.17487/RFC6960, June 2013, <<https://www.rfc-editor.org/info/rfc6960>>.
- [RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <<https://www.rfc-editor.org/info/rfc6962>>.
- [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>>.



- [RFC7540] 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/info/rfc7540>>.
- [RFC9001] Thomson, M., Ed. and S. Turner, Ed., "Using TLS to Secure QUIC", RFC 9001, DOI 10.17487/RFC9001, May 2021, <<https://www.rfc-editor.org/info/rfc9001>>.
- [SIGMAC] Krawczyk, H., "A Unilateral-to-Mutual Authentication Compiler for Key Exchange (with Applications to Client Authentication in TLS 1.3)", 2016, <<https://eprint.iacr.org/2016/711.pdf>>.

## Author's Address

Nick Sullivan  
Cloudflare Inc.  
Email: [nick@cloudflare.com](mailto:nick@cloudflare.com)

tls  
Internet-Draft  
Intended status: Standards Track  
Expires: 24 October 2022

D. Benjamin  
Google, LLC.  
C.A. Wood  
Cloudflare  
22 April 2022

Importing External PSKs for TLS  
draft-ietf-tls-external-psk-importer-08

Abstract

This document describes an interface for importing external Pre-Shared Keys (PSKs) into TLS 1.3.

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-external-psk-importer>.

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 24 October 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
3. Terminology . . . . .	3
4. Overview . . . . .	4
5. PSK Import . . . . .	4
5.1. External PSK Diversification . . . . .	4
5.2. Binder Key Derivation . . . . .	6
6. Deprecating Hash Functions . . . . .	7
7. Incremental Deployment . . . . .	7
8. Security Considerations . . . . .	8
9. Privacy Considerations . . . . .	9
10. IANA Considerations . . . . .	9
11. References . . . . .	10
11.1. Normative References . . . . .	10
11.2. Informative References . . . . .	10
Appendix A. Acknowledgements . . . . .	12
Appendix B. Addressing Selfie . . . . .	12
Authors' Addresses . . . . .	12

## 1. Introduction

TLS 1.3 [RFC8446] supports Pre-Shared Key (PSK) authentication, wherein PSKs can be established via session tickets from prior connections or externally via some out-of-band mechanism. The protocol mandates that each PSK only be used with a single hash function. This was done to simplify protocol analysis. TLS 1.2 [RFC5246], in contrast, has no such requirement, as a PSK may be used with any hash algorithm and the TLS 1.2 pseudorandom function (PRF). While there is no known way in which the same external PSK might produce related output in TLS 1.3 and prior versions, only limited analysis has been done. Applications SHOULD provision separate PSKs for (D)TLS 1.3 and prior versions. In cases where this is not possible, e.g., there are already deployed external PSKs or provisioning is otherwise limited, re-using external PSKs across different versions of TLS may produce related outputs, which may in turn lead to security problems; see [RFC8446], Section E.7.

To mitigate against such problems, this document specifies a PSK Importer interface by which external PSKs may be imported and subsequently bound to a specific key derivation function (KDF) and hash function for use in TLS 1.3 [RFC8446] and DTLS 1.3 [DTLS13]. In

particular, it describes a mechanism for importing PSKs derived from external PSKs by including the target KDF, (D)TLS protocol version, and an optional context string to ensure uniqueness. This process yields a set of candidate PSKs, each of which are bound to a target KDF and protocol, that are separate from those used in (D)TLS 1.2 and prior versions. This expands what would normally have been a single PSK and identity into a set of PSKs and identities.

## 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.

## 3. Terminology

The following terms are used throughout this document:

- \* External PSK (EPSK): A PSK established or provisioned out-of-band (i.e., not from a TLS connection) which is a tuple of (Base Key, External Identity, Hash).
- \* Base Key: The secret value of an EPSK.
- \* External Identity: A sequence of bytes used to identify an EPSK.
- \* Target protocol: The protocol for which a PSK is imported for use.
- \* Target KDF: The KDF for which a PSK is imported for use.
- \* Imported PSK (IPSK): A TLS PSK derived from an EPSK, optional context string, target protocol, and target KDF.
- \* Non-imported PSK: An EPSK which used directly as a TLS PSK without being imported.
- \* Imported Identity: A sequence of bytes used to identify an IPSK.

This document uses presentation language from [RFC8446], Section 3.

## 4. Overview

The PSK Importer interface mirrors that of the TLS Exporters interface (see [RFC8446]) in that it diversifies a key based on some contextual information. In contrast to the Exporters interface, wherein output uniqueness is achieved via an explicit label and context string, the PSK Importer interface defined herein takes an external PSK and identity and "imports" it into TLS, creating a set of "derived" PSKs and identities that are each unique. Each of these derived PSKs are bound to a target protocol, KDF identifier, and optional context string. Additionally, the resulting PSK binder keys are modified with a new derivation label to prevent confusion with non-imported PSKs. Through this interface, importing external PSKs with different identities yields distinct PSK binder keys.

Imported keys do not require negotiation for use since a client and server will not agree upon identities if imported incorrectly. Endpoints may incrementally deploy PSK Importer support by offering non-imported PSKs for TLS versions prior to TLS 1.3. Non-imported and imported PSKs are distinct since their identities are different. See Section 7 for more details.

Endpoints which import external keys MUST NOT use the keys that are input to the import process for any purpose other than the importer, and MUST NOT use the derived keys for any purpose other than TLS PSKs. Moreover, each external PSK fed to the importer process MUST be associated with at most one hash function. This is analogous to the rules in Section 4.2.11 of [RFC8446]. See Section 8 for more discussion.

## 5. PSK Import

This section describes the PSK Importer interface and its underlying diversification mechanism and binder key computation modification.

### 5.1. External PSK Diversification

The PSK Importer interface takes as input an EPSK with External Identity `external_identity` and base key `epsk`, as defined in Section 3, along with an optional context, and transforms it into a set of PSKs and imported identities for use in a connection based on target protocols and KDFs. In particular, for each supported target protocol `target_protocol` and KDF `target_kdf`, the importer constructs an `ImportedIdentity` structure as follows:

```
struct {  
    opaque external_identity<1...2^16-1>;  
    opaque context<0..2^16-1>;  
    uint16 target_protocol;  
    uint16 target_kdf;  
} ImportedIdentity;
```

The list of `ImportedIdentity.target_kdf` values is maintained by IANA as described in Section 10. External PSKs MUST NOT be imported for (D)TLS 1.2 or prior versions. See Section 7 for discussion on how imported PSKs for TLS 1.3 and non-imported PSKs for earlier versions co-exist for incremental deployment.

`ImportedIdentity.context` MUST include the context used to determine the EPSK, if any exists. For example, `ImportedIdentity.context` may include information about peer roles or identities to mitigate Selfie-style reflection attacks [Selfie]. See Appendix B for more details. Since the EPSK is a key derived from an external protocol or sequence of protocols, `ImportedIdentity.context` MUST include a channel binding for the deriving protocols [RFC5056]. The details of this binding are protocol specific and out of scope for this document.

`ImportedIdentity.target_protocol` MUST be the (D)TLS protocol version for which the PSK is being imported. For example, TLS 1.3 [RFC8446] uses 0x0304, which will therefore also be used by QUICv1 [QUIC]. Note that this means the number of PSKs derived from an EPSK is a function of the number of target protocols.

Given an `ImportedIdentity` and corresponding EPSK with base key `epsk`, an Imported PSK `IPSK` with base key `ipskx` is computed as follows:

```
epskx = HKDF-Extract(0, epsk)  
ipskx = HKDF-Expand-Label(epskx, "derived psk",  
                          Hash(ImportedIdentity), L)
```

`L` corresponds to the KDF output length of `ImportedIdentity.target_kdf` as defined in Section 10. For hash-based KDFs, such as `HKDF_SHA256(0x0001)`, this is the length of the hash function output, e.g., 32 octets for SHA256. This is required for the IPSK to be of length suitable for supported ciphersuites. Internally, `HKDF-Expand-Label` uses a label corresponding to `ImportedIdentity.target_protocol`, e.g., "tls13" for TLS 1.3, as per [RFC8446], Section 7.1, or "dtls13" for DTLS 1.3, as per [I-D.ietf-tls-dtls13], Section 5.10.

The identity of `ipskx` as sent on the wire is `ImportedIdentity`, i.e., the serialized content of `ImportedIdentity` is used as the content of `PskIdentity.identity` in the PSK extension. The corresponding PSK input for the TLS 1.3 key schedule is `'ipskx'`.

As the maximum size of the PSK extension is  $2^{16} - 1$  octets, an `Imported Identity` that exceeds this size is likely to cause a decoding error. Therefore, the PSK Importer interface **SHOULD** reject any `ImportedIdentity` that exceeds this size.

The hash function used for HKDF [RFC5869] is that which is associated with the EPSK. It is not the hash function associated with `ImportedIdentity.target_kdf`. If the EPSK does not have such an associated hash function, SHA-256 [SHA2] **SHOULD** be used. Diversifying EPSK by `ImportedIdentity.target_kdf` ensures that an IPSK is only used as input keying material to at most one KDF, thus satisfying the requirements in [RFC8446]. See Section 8 for more details.

Endpoints **SHOULD** generate a compatible `ipskx` for each target ciphersuite they offer. For example, importing a key for `TLS_AES_128_GCM_SHA256` and `TLS_AES_256_GCM_SHA384` would yield two PSKs, one for HKDF-SHA256 and another for HKDF-SHA384. In contrast, if `TLS_AES_128_GCM_SHA256` and `TLS_CHACHA20_POLY1305_SHA256` are supported, only one derived key is necessary. Each ciphersuite uniquely identifies the target KDF. Future specifications that change the way the KDF is negotiated will need to update this specification to make clear how target KDFs are determined for the import process.

EPSKs **MAY** be imported before the start of a connection if the target KDFs, protocols, and context string(s) are known a priori. EPSKs **MAY** also be imported for early data use if they are bound to the protocol settings and configuration that are required for sending early data. Minimally, this means that the Application-Layer Protocol Negotiation value [RFC7301], QUIC transport parameters (if used for QUIC), and any other relevant parameters that are negotiated for early data **MUST** be provisioned alongside these EPSKs.

## 5.2. Binder Key Derivation

To prevent confusion between imported and non-imported PSKs, imported PSKs change the PSK binder key derivation label. In particular, the standard TLS 1.3 PSK binder key computation is defined as follows:

```

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

```

Imported PSKs use the string "imp binder" rather than "ext binder" or "res binder" when deriving binder\_key. This means the binder key is computed as follows:

```

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

```

This new label ensures a client and server will negotiate use of an external PSK if and only if (a) both endpoints import the PSK or (b) neither endpoint imports the PSK. As binder\_key is a leaf key, changing its computation does not affect any other key.

## 6. Deprecating Hash Functions

If a client or server wishes to deprecate a hash function and no longer use it for TLS 1.3, they remove the corresponding KDF from the set of target KDFs used for importing keys. This does not affect the KDF operation used to derive Imported PSKs.

## 7. Incremental Deployment

In deployments that already have PSKs provisioned and in use with TLS 1.2, attempting to incrementally deploy the importer mechanism would then result in concurrent use of the already provisioned PSK both directly as a TLS 1.2 PSK and as an EPSK, which in turn could mean that the same KDF and key would be used in two different protocol contexts. This is not a recommended configuration; see Section 8 for more details. However, the benefits of using TLS 1.3 and of using PSK importers may prove sufficiently compelling that existing deployments choose to enable this noncompliant configuration for a brief transition period while new software (using TLS 1.3 and



importers) is deployed. Operators are advised to make any such transition period as short as possible.

## 8. Security Considerations

The PSK Importer security goals can be roughly stated as follows: avoid PSK re-use across KDFs while properly authenticating endpoints. When modeled as computational extractors, KDFs assume that input keying material (IKM) is sampled from some "source" probability distribution and that any two IKM values are chosen independently of each other [Kraw10]. This source-independence requirement implies that the same IKM value cannot be used for two different KDFs.

PSK-based authentication is functionally equivalent to session resumption in that a connection uses existing key material to authenticate both endpoints. Following the work of [BAA15], this is a form of compound authentication. Loosely speaking, compound authentication is the property that an execution of multiple authentication protocols, wherein at least one is uncompromised, jointly authenticates all protocols. Authenticating with an externally provisioned PSK, therefore, should ideally authenticate both the TLS connection and the external provisioning process. Typically, the external provision process produces a PSK and corresponding context from which the PSK was derived and in which it should be used. If available, this is used as the `ImportedIdentity.context` value. We refer to an external PSK without such context as "context-free".

Thus, in considering the source-independence and compound authentication requirements, the PSK Import interface described in this document aims to achieve the following goals:

1. Externally provisioned PSKs imported into a TLS connection achieve compound authentication of the provisioning process and connection.
2. Context-free PSKs only achieve authentication within the context of a single connection.
3. Imported PSKs are not used as IKM for two different KDFs.
4. Imported PSKs do not collide with future protocol versions and KDFs.

There are no known related outputs or security issues caused from the process for computing Imported PSKs from an external PSK and the processing of existing external PSKs used in (D)TLS 1.2 and below, as noted in Section 7. However, only limited analysis has been done,

which is an additional reason why applications SHOULD provision separate PSKs for (D)TLS 1.3 and prior versions, even when the importer interface is used in (D)TLS 1.3.

The PSK Importer does not prevent applications from constructing non-importer PSK identities that collide with imported PSK identities.

## 9. Privacy Considerations

External PSK identities are commonly static by design so that endpoints may use them to lookup keying material. As a result, for some systems and use cases, this identity may become a persistent tracking identifier.

Note also that `ImportedIdentity.context` is visible in cleartext on the wire as part of the PSK identity. Unless otherwise protected by a mechanism such as TLS Encrypted ClientHello [ECH], applications SHOULD NOT put sensitive information in this field.

## 10. IANA Considerations

This specification introduces a new registry for TLS KDF identifiers, titled "TLS KDF Identifiers", under the existing "Transport Layer Security (TLS) Parameters" heading.

The entries in the registry are:

KDF Description	Value	Reference
Reserved	0x0000	N/A
HKDF_SHA256	0x0001	[RFC5869]
HKDF_SHA384	0x0002	[RFC5869]

Table 1: Target KDF Registry

New target KDF values are allocated according to the following process:

- \* Values in the range 0x0000-0xfeff are assigned via Specification Required [RFC8126].
- \* Values in the range 0xff00-0xffff are reserved for Private Use [RFC8126].

The procedures for requesting values in the Specification Required space are specified in Section 17 of [RFC8447].

## 11. References

### 11.1. Normative References

- [DTLS13] 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://www.ietf.org/archive/id/draft-ietf-tls-dtls13-43.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>>.
- [RFC5056] Williams, N., "On the Use of Channel Bindings to Secure Channels", RFC 5056, DOI 10.17487/RFC5056, November 2007, <<https://www.rfc-editor.org/info/rfc5056>>.
- [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>>.
- [RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/info/rfc8126>>.
- [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>>.
- [RFC8447] Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", RFC 8447, DOI 10.17487/RFC8447, August 2018, <<https://www.rfc-editor.org/info/rfc8447>>.

### 11.2. Informative References

- [BAA15] Bhargavan, K., Delignat-Lavaud, A., and A. Pironti, "Verified Contributive Channel Bindings for Compound Authentication", DOI 10.14722/ndss.2015.23277, Proceedings 2015 Network and Distributed System Security Symposium, 2015, <<https://doi.org/10.14722/ndss.2015.23277>>.
- [ECH] 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://www.ietf.org/archive/id/draft-ietf-tls-esni-14.txt>>.
- [I-D.ietf-tls-dtls13] 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://www.ietf.org/archive/id/draft-ietf-tls-dtls13-43.txt>>.
- [Kraw10] Krawczyk, H., "Cryptographic Extraction and Key Derivation: The HKDF Scheme", Proceedings of CRYPTO 2010 , 2010, <<https://eprint.iacr.org/2010/264>>.
- [QUIC] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <<https://www.rfc-editor.org/info/rfc9000>>.
- [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>>.
- [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>>.
- [Selfie] Drucker, N. and S. Gueron, "Selfie: reflections on TLS 1.3 with PSK", 2019, <<https://eprint.iacr.org/2019/347.pdf>>.
- [SHA2] National Institute of Standards and Technology, "Secure Hash Standard", FIPS PUB 180-3 , October 2008.

## Appendix A. Acknowledgements

The authors thank Eric Rescorla and Martin Thomson for discussions that led to the production of this document, as well as Christian Huitema for input regarding privacy considerations of external PSKs. John Mattsson provided input regarding PSK importer deployment considerations. Hugo Krawczyk provided guidance for the security considerations. Martin Thomson, Jonathan Hoyland, Scott Hollenbeck, Benjamin Kaduk, and others all provided reviews, feedback, and suggestions for improving the document.

## Appendix B. Addressing Selfie

The Selfie attack [Selfie] relies on a misuse of the PSK interface. The PSK interface makes the implicit assumption that each PSK is known only to one client and one server. If multiple clients or multiple servers with distinct roles share a PSK, TLS only authenticates the entire group. A node successfully authenticates its peer as being in the group whether the peer is another node or itself. Note that this case can also occur when there are two nodes sharing a PSK without predetermined roles.

Applications which require authenticating finer-grained roles while still configuring a single shared PSK across all nodes can resolve this mismatch either by exchanging roles over the TLS connection after the handshake or by incorporating the roles of both the client and server into the IPSK context string. For instance, if an application identifies each node by MAC address, it could use the following context string.

```
struct {  
    opaque client_mac<0..2^8-1>;  
    opaque server_mac<0..2^8-1>;  
} Context;
```

If an attacker then redirects a ClientHello intended for one node to a different node, including the node that generated the ClientHello, the receiver will compute a different context string and the handshake will not complete.

Note that, in this scenario, there is still a single shared PSK across all nodes, so each node must be trusted not to impersonate another node's role.

## Authors' Addresses

David Benjamin  
Google, LLC.

Email: davidben@google.com

Christopher A. Wood

Cloudflare

Email: caw@heapingbits.net

Internet Engineering Task Force  
Internet-Draft  
Updates: 5246 (if approved)  
Intended status: Standards Track  
Expires: 24 March 2022

L.V. Velvindron  
cyberstorm.mu  
K.M. Moriarty  
CIS  
A.G. Ghedini  
Cloudflare Inc.  
20 September 2021

Deprecating MD5 and SHA-1 signature hashes in (D)TLS 1.2  
draft-ietf-tls-md5-sha1-deprecate-09

Abstract

The MD5 and SHA-1 hashing algorithms are increasingly vulnerable to attack and this document deprecates their use in TLS 1.2 digital signatures. However, this document does not deprecate SHA-1 in HMAC for record protection. This document updates RFC 5246.

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 24 March 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. Requirements Language . . . . .	3
2. Signature Algorithms . . . . .	3
3. Certificate Request . . . . .	3
4. Server Key Exchange . . . . .	3
5. Certificate Verify . . . . .	3
6. IANA Considerations . . . . .	3
7. Security Considerations . . . . .	4
8. Acknowledgement . . . . .	4
9. References . . . . .	4
9.1. Normative References . . . . .	4
9.2. Informative References . . . . .	5
Authors' Addresses . . . . .	5

## 1. Introduction

The usage of MD5 and SHA-1 for signature hashing in TLS 1.2 is specified in [RFC5246]. MD5 and SHA-1 have been proven to be insecure, subject to collision attacks [Wang]. In 2011, [RFC6151] detailed the security considerations, including collision attacks for MD5. NIST formally deprecated use of SHA-1 in 2011 [NISTSP800-131A-R2] and disallowed its use for digital signatures at the end of 2013, based on both the Wang et al. attack and the potential for brute-force attack. In 2016, researchers from INRIA identified a new class of transcript collision attacks on TLS (and other protocols) that rely on efficient collision-finding algorithms on the underlying hash constructions [Transcript-Collision]. Further, in 2017, researchers from Google and CWI Amsterdam [SHA-1-Collision] proved SHA-1 collision attacks were practical. This document updates [RFC5246] in such a way that MD5 and SHA-1 MUST NOT be used for digital signatures. However, this document does not deprecate SHA-1 in HMAC for record protection. Note that the CABF has also deprecated use of SHA-1 for use in certificate signatures [CABF].



### 1.1. Requirements Language

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. Signature Algorithms

Clients MUST include the `signature_algorithms` extension. Clients MUST NOT include MD5 and SHA-1 in this extension.

## 3. Certificate Request

Servers SHOULD NOT include MD5 and SHA-1 in `CertificateRequest` messages.

## 4. Server Key Exchange

Servers MUST NOT include MD5 and SHA-1 in `ServerKeyExchange` messages. If the client receives a `ServerKeyExchange` message indicating MD5 or SHA-1, then it MUST abort the connection with an `illegal_parameter` alert.

## 5. Certificate Verify

Clients MUST NOT include MD5 and SHA-1 in `CertificateVerify` messages. If a server receives a `CertificateVerify` message with MD5 or SHA-1 it MUST abort the connection with an `illegal_parameter` alert.

## 6. IANA Considerations

The document updates the "TLS SignatureScheme" registry to change the recommended status of SHA-1 based signature schemes to N (not recommended) as defined by [RFC8447]. The following entries are to be updated:

Value	Description	Recommended	Reference
0x0201	rsa_pkcs1_sha1	N	[RFC8446] [RFCTBD]
0x0203	ecdsa_sha1	N	[RFC8446] [RFCTBD]

Table 1

Other entries of the registry remain the same.

IANA is also requested to update the Reference for the TLS SignatureAlgorithm and TLS HashAlgorithm registries to refer to this RFC:

OLD:

Reference

[RFC5246] [RFC8447]

NEW:

Reference

[RFC5246] [RFC8447] [RFC-to-be]

## 7. Security Considerations

Concerns with TLS 1.2 implementations falling back to SHA-1 is an issue. This document updates the TLS 1.2 specification to deprecate support for MD5 and SHA-1 for digital signatures. However, this document does not deprecate SHA-1 in HMAC for record protection.

## 8. Acknowledgement

The authors would like to thank Hubert Kario for his help in writing the initial draft. We are also grateful to Daniel Migault, Martin Thomson, Sean Turner, Christopher Wood and David Cooper for their feedback.

## 9. References

### 9.1. 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>>.

[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>>.

- [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>>.
- [RFC8447] Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", RFC 8447, DOI 10.17487/RFC8447, August 2018, <<https://www.rfc-editor.org/info/rfc8447>>.

## 9.2. Informative References

- [CABF] CA/Browser Forum, "Ballot 118 -- SHA-1 Sunset (passed)", 2014, <<https://cabforum.org/2014/10/16/ballot-118-sha-1-sunset/>>.
- [NISTSP800-131A-R2] Barker, E.B. and A.R. Roginsky, "Transitioning the Use of Cryptographic Algorithms and Key Lengths", March 2019, <<https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar2.pdf>>.
- [RFC6151] Turner, S. and L. Chen, "Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms", RFC 6151, DOI 10.17487/RFC6151, March 2011, <<https://www.rfc-editor.org/info/rfc6151>>.
- [SHA-1-Collision] Stevens, M.S., Bursztein, E.B., Karpman, P.K., Albertini, A.A., and Y.M. Markov, "The first collision for full SHA-1", March 2019, <<https://eprint.iacr.org/2017/190>>.
- [Transcript-Collision] Bhargavan, K.B. and G.L. Leurent, "Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH", February 2016, <<https://hal.inria.fr/hal-01244855/document>>.
- [Wang] Wang, X.W., Yin, Y.Y., and H.Y. Yu, "Finding Collisions in the Full SHA-1", 2005, <<https://www.iacr.org/archive/crypto2005/36210017/36210017.pdf>>.

Authors' Addresses

Loganaden Velvindron  
cyberstorm.mu  
Rose Hill  
Mauritius

Phone: +230 59762817  
Email: logan@cyberstorm.mu

Kathleen Moriarty  
Center for Internet Security  
East Greenbush, NY  
United States of America

Email: Kathleen.Moriarty.ietf@gmail.com

Alessandro Ghedini  
Cloudflare Inc.

Email: alessandro@cloudflare.com

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

R. Barnes  
Cisco  
S. Iyengar  
Facebook  
N. Sullivan  
Cloudflare  
E. Rescorla  
Mozilla  
9 May 2022

Delegated Credentials for (D)TLS  
draft-ietf-tls-subcerts-13

## Abstract

The organizational separation between operators of TLS and DTLS endpoints and the certification authority can create limitations. For example, the lifetime of certificates, how they may be used, and the algorithms they support are ultimately determined by the certification authority. This document describes a mechanism to to overcome some of these limitations by enabling operators to delegate their own credentials for use in TLS and DTLS without breaking compatibility with peers that do not support this specification.

## 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/tls-subcerts> (<https://github.com/tlswg/tls-subcerts>).

## 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 10 November 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. Conventions and Terminology
  - 2.1. Change Log
- 3. Solution Overview
  - 3.1. Rationale
  - 3.2. Related Work
- 4. Delegated Credentials
  - 4.1. Client and Server Behavior
    - 4.1.1. Server Authentication
    - 4.1.2. Client Authentication
    - 4.1.3. Validating a Delegated Credential
  - 4.2. Certificate Requirements
- 5. Operational Considerations
  - 5.1. Client Clock Skew
- 6. IANA Considerations
- 7. Security Considerations
  - 7.1. Security of Delegated Credential's Private Key
  - 7.2. Re-use of Delegated Credentials in Multiple Contexts
  - 7.3. Revocation of Delegated Credentials
  - 7.4. Interactions with Session Resumption
  - 7.5. Privacy Considerations
  - 7.6. The Impact of Signature Forgery Attacks
- 8. Acknowledgements
- 9. References
  - 9.1. Normative References
  - 9.2. Informative References
- Appendix A. ASN.1 Module
- Appendix B. Example Certificate
- Authors' Addresses

## 1. Introduction

Server operators often deploy (D)TLS termination services in locations such as remote data centers or Content Delivery Networks (CDNs) where it may be difficult to detect compromises of private key material corresponding to TLS certificates. Short-lived certificates may be used to limit the exposure of keys in these cases.

However, short-lived certificates need to be renewed more frequently than long-lived certificates. If an external Certification Authority (CA) is unable to issue a certificate in time to replace a deployed certificate, the server would no longer be able to present a valid certificate to clients. With short-lived certificates, there is a smaller window of time to renew a certificates and therefore a higher risk that an outage at a CA will negatively affect the uptime of the TLS-fronted service.

Typically, a (D)TLS server uses a certificate provided by some entity other than the operator of the server (a CA) [RFC8446] [RFC5280]. This organizational separation makes the (D)TLS server operator dependent on the CA for some aspects of its operations, for example:

- \* Whenever the server operator wants to deploy a new certificate, it has to interact with the CA.

- \* The CA might only issue credentials containing certain types of public key, which can limit the set of (D)TLS signature schemes usable by the server operator.

To reduce the dependency on external CAs, this document specifies a limited delegation mechanism that allows a (D)TLS peer to issue its own credentials within the scope of a certificate issued by an external CA. These credentials only enable the recipient of the delegation to speak for names that the CA has authorized. Furthermore, this mechanism allows the server to use modern signature algorithms such as Ed25519 [RFC8032] even if their CA does not support them.

This document refers to the certificate issued by the CA as a "certificate", or "delegation certificate", and the one issued by the operator as a "delegated credential" or "DC".

## 2. Conventions and 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.

### 2.1. Change Log

RFC EDITOR PLEASE DELETE THIS SECTION.

(\*) indicates changes to the wire protocol.

draft-11

- \* Editorial changes based on AD comments
- \* Add support for DTLs
- \* Address address ambiguity in cert expiry

draft-10

- \* Address superficial comments
- \* Add example certificate

draft-09

- \* Address case nits
- \* Fix section bullets in 4.1.3.
- \* Add operational considerations section for clock skew
- \* Add text around using an oracle to forge DCs in the future and past
- \* Add text about certificate extension vs ECU

draft-08

- \* Include details about the impact of signature forgery attacks

- \* Copy edits
- \* Fix section about DC reuse
- \* Incorporate feedback from Jonathan Hammell and Kevin Jacobs on the list

draft-07

- \* Minor text improvements

draft-06

- \* Modified IANA section, fixed nits

draft-05

- \* Removed support for PKCS 1.5 RSA signature algorithms.
- \* Additional security considerations.

draft-04

- \* Add support for client certificates.

draft-03

- \* Remove protocol version from the Credential structure. (\*)

draft-02

- \* Change public key type. (\*)
- \* Change DelegationUsage extension to be NULL and define its object identifier.
- \* Drop support for TLS 1.2.
- \* Add the protocol version and credential signature algorithm to the Credential structure. (\*)
- \* Specify undefined behavior in a few cases: when the client receives a DC without indicated support; when the client indicates the extension in an invalid protocol version; and when DCs are sent as extensions to certificates other than the end-entity certificate.

### 3. Solution Overview

A delegated credential (DC) is a digitally signed data structure with two semantic fields: a validity interval and a public key (along with its associated signature algorithm). The signature on the delegated credential indicates a delegation from the certificate that is issued to the peer. The private key used to sign a credential corresponds to the public key of the peer's X.509 end-entity certificate [RFC5280].

A (D)TLS handshake that uses delegated credentials differs from a standard handshake in a few important ways:

- \* The initiating peer provides an extension in its ClientHello or



CertificateRequest that indicates support for this mechanism.

- \* The peer sending the Certificate message provides both the certificate chain terminating in its certificate as well as the delegated credential.
- \* The initiator uses information from the peer's certificate to verify the delegated credential and that the peer is asserting an expected identity, determining an authentication result for the peer.
- \* Peers accepting the delegated credential use it as the certificate key for the (D)TLS handshake.

As detailed in Section 4, the delegated credential is cryptographically bound to the end-entity certificate with which the credential may be used. This document specifies the use of delegated credentials in (D)TLS 1.3 or later; their use in prior versions of the protocol is not allowed.

Delegated credentials allow a peer to terminate (D)TLS connections on behalf of the certificate owner. If a credential is stolen, there is no mechanism for revoking it without revoking the certificate itself. To limit exposure in case of the compromise of a delegated credential's private key, delegated credentials have a maximum validity period. In the absence of an application profile standard specifying otherwise, the maximum validity period is set to 7 days. Peers MUST NOT issue credentials with a validity period longer than the maximum validity period or that extends beyond the validity period of the delegation certificate. This mechanism is described in detail in Section 4.1.

It was noted in [XPROT] that certificates in use by servers that support outdated protocols such as SSLv2 can be used to forge signatures for certificates that contain the keyEncipherment KeyUsage ([RFC5280] section 4.2.1.3). In order to reduce the risk of cross-protocol attacks on certificates that are not intended to be used with DC-capable TLS stacks, we define a new DelegationUsage extension to X.509 that permits use of delegated credentials. (See Section 4.2.)

### 3.1. Rationale

Delegated credentials present a better alternative than other delegation mechanisms like proxy certificates [RFC3820] for several reasons:

- \* There is no change needed to certificate validation at the PKI layer.
- \* X.509 semantics are very rich. This can cause unintended consequences if a service owner creates a proxy certificate where the properties differ from the leaf certificate. Proxy certificates can be useful in controlled environments, but remain a risk in scenarios where the additional flexibility they provide is not necessary. For this reason, delegated credentials have very restricted semantics that should not conflict with X.509 semantics.
- \* Proxy certificates rely on the certificate path building process to establish a binding between the proxy certificate and the end-entity certificate. Since the certificate path building process

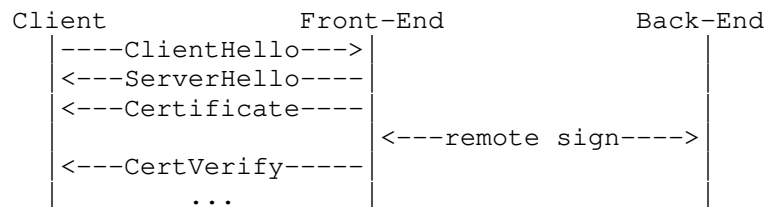
is not cryptographically protected, it is possible that a proxy certificate could be bound to another certificate with the same public key, with different X.509 parameters. Delegated credentials, which rely on a cryptographic binding between the entire certificate and the delegated credential, cannot.

- \* Each delegated credential is bound to a specific signature algorithm for use in the (D)TLS handshake ([RFC8446] section 4.2.3). This prevents them from being used with other, perhaps unintended, signature algorithms. The signature algorithm bound to the delegated credential can be chosen independently of the set of signature algorithms supported by the end-entity certificate.

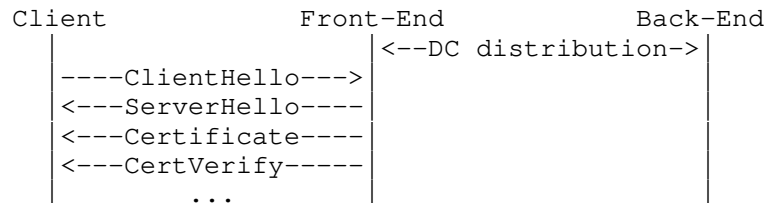
### 3.2. Related Work

Many of the use cases for delegated credentials can also be addressed using purely server-side mechanisms that do not require changes to client behavior (e.g., a PKCS#11 interface or a remote signing mechanism, [KEYLESS] being one example). These mechanisms, however, incur per-transaction latency, since the front-end server has to interact with a back-end server that holds a private key. The mechanism proposed in this document allows the delegation to be done off-line, with no per-transaction latency. The figure below compares the message flows for these two mechanisms with (D)TLS 1.3 [RFC8446] [I-D.ietf-tls-dtls13].

Remote key signing:



Delegated Credential:



These two mechanisms can be complementary. A server could use delegated credentials for clients that support them, while using a server-side mechanism to support legacy clients. Both mechanisms require a trusted relationship between the Front-End and Back-End -- the delegated credential can be used in place of a certificate private key.

Use of short-lived certificates with automated certificate issuance, e.g., with Automated Certificate Management Environment (ACME) [RFC8555], reduces the risk of key compromise, but has several limitations. Specifically, it introduces an operationally-critical dependency on an external party (the CA). It also limits the types of algorithms supported for (D)TLS authentication to those the CA is willing to issue a certificate for. Nonetheless, existing automated issuance APIs like ACME may be useful for provisioning delegated

credentials.

#### 4. Delegated Credentials

While X.509 forbids end-entity certificates from being used as issuers for other certificates, it is valid to use them to issue other signed objects as long as the certificate contains the digitalSignature KeyUsage ([RFC5280] section 4.2.1.3). (All certificates compatible with TLS 1.3 are required to contain the digitalSignature KeyUsage.) We define a new signed object format that would encode only the semantics that are needed for this application. The Credential has the following structure:

```
struct {
    uint32 valid_time;
    SignatureScheme expected_cert_verify_algorithm;
    opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;
} Credential;
```

**valid\_time:** Time, in seconds relative to the delegation certificate's notBefore value, after which the delegated credential is no longer valid. Endpoints will reject delegated credentials that expire more than 7 days from the current time (as described in Section 4.1) based on the default (see Section 3).

**expected\_cert\_verify\_algorithm:** The signature algorithm of the Credential key pair, where the type SignatureScheme is as defined in [RFC8446]. This is expected to be the same as the sender's CertificateVerify.algorithm (as described in Section 4.1.3). Only signature algorithms allowed for use in CertificateVerify messages are allowed. When using RSA, the public key MUST NOT use the rsaEncryption OID. As a result, the following algorithms are not allowed for use with delegated credentials: rsa\_pss\_rsae\_sha256, rsa\_pss\_rsae\_sha384, rsa\_pss\_rsae\_sha512.

**ASN1\_subjectPublicKeyInfo:** The Credential's public key, a DER-encoded [X.690] SubjectPublicKeyInfo as defined in [RFC5280].

The DelegatedCredential has the following structure:

```
struct {
    Credential cred;
    SignatureScheme algorithm;
    opaque signature<0..2^16-1>;
} DelegatedCredential;
```

**cred:** The Credential structure as previously defined.

**algorithm:** The signature algorithm used to verify DelegatedCredential.signature.

**signature:** The delegation, a signature that binds the credential to the end-entity certificate's public key as specified below. The signature scheme is specified by DelegatedCredential.algorithm.

The signature of the DelegatedCredential is computed over the concatenation of:

1. A string that consists of octet 32 (0x20) repeated 64 times.
2. The context string "TLS, server delegated credentials" for server authentication and "TLS, client delegated credentials" for client

authentication.

3. A single 0 byte, which serves as the separator.
4. The DER-encoded X.509 end-entity certificate used to sign the DelegatedCredential.
5. DelegatedCredential.cred.
6. DelegatedCredential.algorithm.

The signature is computed by using the private key of the peer's end-entity certificate, with the algorithm indicated by DelegatedCredential.algorithm.

The signature effectively binds the credential to the parameters of the handshake in which it is used. In particular, it ensures that credentials are only used with the certificate and signature algorithm chosen by the delegator.

The code changes required in order to create and verify delegated credentials, and the implementation complexity this entails, are localized to the (D)TLS stack. This has the advantage of avoiding changes to the often-delicate security-critical PKI code.

#### 4.1. Client and Server Behavior

This document defines the following (D)TLS extension code point.

```
enum {  
    ...  
    delegated_credential(34),  
    (65535)  
} ExtensionType;
```

##### 4.1.1. Server Authentication

A client which supports this specification SHALL send a "delegated\_credential" extension in its ClientHello. The body of the extension consists of a SignatureSchemeList (defined in [RFC8446]):

```
struct {  
    SignatureScheme supported_signature_algorithm<2..2^16-2>;  
} SignatureSchemeList;
```

If the client receives a delegated credential without having indicated support in its ClientHello, then the client MUST abort the handshake with an "unexpected\_message" alert.

If the extension is present, the server MAY send a delegated credential; if the extension is not present, the server MUST NOT send a delegated credential. The server MUST ignore the extension unless (D)TLS 1.3 or a later version is negotiated. An example of when a server could choose not to send a delegated credential is when the SignatureSchemes listed only contain signature schemes for which a corresponding delegated credential does not exist or are otherwise unsuitable for the connection.

The server MUST send the delegated credential as an extension in the CertificateEntry of its end-entity certificate; the client SHOULD ignore delegated credentials sent as extensions to any other certificate.

The algorithm field MUST be of a type advertised by the client in the "signature\_algorithms" extension of the ClientHello message and the expected\_cert\_verify\_algorithm field MUST be of a type advertised by the client in the SignatureSchemeList and is considered invalid otherwise. Clients that receive invalid delegated credentials MUST terminate the connection with an "illegal\_parameter" alert.

#### 4.1.2. Client Authentication

A server that supports this specification SHALL send a "delegated\_credential" extension in the CertificateRequest message when requesting client authentication. The body of the extension consists of a SignatureSchemeList. If the server receives a delegated credential without having indicated support in its CertificateRequest, then the server MUST abort with an "unexpected\_message" alert.

If the extension is present, the client MAY send a delegated credential; if the extension is not present, the client MUST NOT send a delegated credential. The client MUST ignore the extension unless (D)TLS 1.3 or a later version is negotiated.

The client MUST send the delegated credential as an extension in the CertificateEntry of its end-entity certificate; the server SHOULD ignore delegated credentials sent as extensions to any other certificate.

The algorithm field MUST be of a type advertised by the server in the "signature\_algorithms" extension of the CertificateRequest message and the expected\_cert\_verify\_algorithm field MUST be of a type advertised by the server in the SignatureSchemeList and is considered invalid otherwise. Servers that receive invalid delegated credentials MUST terminate the connection with an "illegal\_parameter" alert.

#### 4.1.3. Validating a Delegated Credential

On receiving a delegated credential and certificate chain, the peer validates the certificate chain and matches the end-entity certificate to the peer's expected identity in the same way that it is done when delegated credentials are not in use. It then performs the following checks with expiry time set to the delegation certificate's notBefore value plus DelegatedCredential.cred.valid\_time:

1. Verify that the current time is within the validity interval of the credential. This is done by asserting that the current time does not exceed the expiry time. (The start time of the credential is implicitly validated as part of certificate validation.)
2. Verify that the delegated credential's remaining validity period is no more than the maximum validity period. This is done by asserting that the expiry time does not exceed the current time plus the maximum validity period (7 days by default).
3. Verify that expected\_cert\_verify\_algorithm matches the scheme indicated in the peer's CertificateVerify message and that the algorithm is allowed for use with delegated credentials.
4. Verify that the end-entity certificate satisfies the conditions

in Section 4.2.

5. Use the public key in the peer's end-entity certificate to verify the signature of the credential using the algorithm indicated by `DelegatedCredential.algorithm`.

If one or more of these checks fail, then the delegated credential is deemed invalid. Clients and servers that receive invalid delegated credentials MUST terminate the connection with an "illegal\_parameter" alert.

If successful, the participant receiving the Certificate message uses the public key in `DelegatedCredential.cred` to verify the signature in the peer's `CertificateVerify` message.

#### 4.2. Certificate Requirements

This document defines a new X.509 extension, `DelegationUsage`, to be used in the certificate when the certificate permits the usage of delegated credentials. What follows is the ASN.1 [X.680] for the `DelegationUsage` certificate extension.

```
ext-delegationUsage EXTENSION ::= {  
    SYNTAX DelegationUsage IDENTIFIED BY id-pe-delegationUsage  
}
```

```
DelegationUsage ::= NULL
```

```
id-pe-delegationUsage OBJECT IDENTIFIER ::=  
    { iso(1) identified-organization(3) dod(6) internet(1)  
      private(4) enterprise(1) id-cloudflare(44363) 44 }
```

The extension MUST be marked non-critical. (See Section 4.2 of [RFC5280].) An endpoint MUST NOT accept a delegated credential unless the peer's end-entity certificate satisfies the following criteria:

- \* It has the `DelegationUsage` extension.
- \* It has the `digitalSignature KeyUsage` (see the `KeyUsage` extension defined in [RFC5280]).

A new extension was chosen instead of adding a new Extended Key Usage (EKU) to be compatible with deployed (D)TLS and PKI software stacks without requiring CAs to issue new intermediate certificates.

#### 5. Operational Considerations

The operational consideration documented in this section should be taken into consideration when using Delegated Certificates.

##### 5.1. Client Clock Skew

One of the risks of deploying a short-lived credential system based on absolute time is client clock skew. If a client's clock is sufficiently ahead or behind of the server's clock, then clients will reject delegated credentials that are valid from the server's perspective. Clock skew also affects the validity of the original certificates. The lifetime of the delegated credential should be set taking clock skew into account. Clock skew may affect a delegated credential at the beginning and end of its validity periods, which should also be taken into account.

## 6. IANA Considerations

This document registers the "delegated\_credential" extension in the "TLS ExtensionType Values" registry. The "delegated\_credential" extension has been assigned a code point of 34. The IANA registry lists this extension as "Recommended" (i.e., "Y") and indicates that it may appear in the ClientHello (CH), CertificateRequest (CR), or Certificate (CT) messages in (D)TLS 1.3 [RFC8446] [I-D.ietf-tls-dtls13]. Additionally, the "DTLS-Only" column is assigned the value "N".

This document also defines an ASN.1 module for the DelegationUsage certificate extension in Appendix A. IANA has registered value 95 for "id-mod-delegated-credential-extn" in the "SMI Security for PKIX Module Identifier" (1.3.5.1.5.5.7.0) registry. An OID for the DelegationUsage certificate extension is not needed as it is already assigned to the extension from Cloudflare's IANA Private Enterprise Number (PEN) arc.

## 7. Security Considerations

The security consideration documented in this section should be taken into consideration when using Delegated Certificates.

### 7.1. Security of Delegated Credential's Private Key

Delegated credentials limit the exposure of the private key used in a (D)TLS connection by limiting its validity period. An attacker who compromises the private key of a delegated credential can impersonate the compromised party in new TLS connections until the delegated credential expires.

However, they cannot create new delegated credentials. Thus, delegated credentials should not be used to send a delegation to an untrusted party, but are meant to be used between parties that have some trust relationship with each other. The secrecy of the delegated credential's private key is thus important and access control mechanisms SHOULD be used to protect it, including file system controls, physical security, or hardware security modules.

### 7.2. Re-use of Delegated Credentials in Multiple Contexts

It is not possible to use the same delegated credential for both client and server authentication because issuing parties compute the corresponding signature using a context string unique to the intended role (client or server).

### 7.3. Revocation of Delegated Credentials

Delegated credentials do not provide any additional form of early revocation. Since it is short lived, the expiry of the delegated credential revokes the credential. Revocation of the long term private key that signs the delegated credential (from the end-entity certificate) also implicitly revokes the delegated credential.

### 7.4. Interactions with Session Resumption

If a peer decides to cache the certificate chain and re-validate it when resuming a connection, they SHOULD also cache the associated delegated credential and re-validate it. Failing to do so may result in resuming connections for which the DC has expired.

## 7.5. Privacy Considerations

Delegated credentials can be valid for 7 days (by default) and it is much easier for a service to create delegated credentials than a certificate signed by a CA. A service could determine the client time and clock skew by creating several delegated credentials with different expiry timestamps and observing whether the client would accept it. Client time could be unique and thus privacy sensitive clients, such as browsers in incognito mode, who do not trust the service might not want to advertise support for delegated credentials or limit the number of probes that a server can perform.

## 7.6. The Impact of Signature Forgery Attacks

Delegated credentials are only used in (D)TLS 1.3 connections. However, the certificate that signs a delegated credential may be used in other contexts such as (D)TLS 1.2. Using a certificate in multiple contexts opens up a potential cross-protocol attack against delegated credentials in (D)TLS 1.3.

When (D)TLS 1.2 servers support RSA key exchange, they may be vulnerable to attacks that allow forging an RSA signature over an arbitrary message [BLEI]. TLS 1.2 [RFC5246] (Section 7.4.7.1.) describes a mitigation strategy requiring careful implementation of timing resistant countermeasures for preventing these attacks. Experience shows that in practice, server implementations may fail to fully stop these attacks due to the complexity of this mitigation [ROBOT]. For (D)TLS 1.2 servers that support RSA key exchange using a DC-enabled end-entity certificate, a hypothetical signature forgery attack would allow forging a signature over a delegated credential. The forged delegated credential could then be used by the attacker as the equivalent of a man-in-the-middle certificate, valid for a maximum of 7 days (if the default `valid_time` is used).

Server operators should therefore minimize the risk of using DC-enabled end-entity certificates where a signature forgery oracle may be present. If possible, server operators may choose to use DC-enabled certificates only for signing credentials, and not for serving non-DC (D)TLS traffic. Furthermore, server operators may use elliptic curve certificates for DC-enabled traffic, while using RSA certificates without the `DelegationUsage` certificate extension for non-DC traffic; this completely prevents such attacks.

Note that if a signature can be forged over an arbitrary credential, the attacker can choose any value for the `valid_time` field. Repeated signature forgeries therefore allow the attacker to create multiple delegated credentials that can cover the entire validity period of the certificate. Temporary exposure of the key or a signing oracle may allow the attacker to impersonate a server for the lifetime of the certificate.

## 8. Acknowledgements

Thanks to David Benjamin, Christopher Patton, Kyle Nekritz, Anirudh Ramachandran, Benjamin Kaduk, Kazuho Oku, Daniel Kahn Gillmor, Watson Ladd, Robert Merget, Juraj Somorovsky, Nimrod Aviram for their discussions, ideas, and bugs they have found.

## 9. References

### 9.1. Normative References



- [I-D.ietf-tls-dtls13] 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>>.
- [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>>.
- [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/rfc/rfc5280>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/rfc/rfc8174>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/rfc/rfc8446>>.
- [X.680] ITU-T, "Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation", ISO/IEC 8824-1:2015, November 2015.
- [X.690] ITU-T, "Information technology - ASN.1 encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ISO/IEC 8825-1:2015, November 2015.

## 9.2. 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.
- [KEYLESS] Sullivan, N. and D. Stebila, "An Analysis of TLS Handshake Proxying", IEEE Trustcom/BigDataSE/ISPA 2015 , 2015.
- [RFC3820] Tuecke, S., Welch, V., Engert, D., Pearlman, L., and M. Thompson, "Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile", RFC 3820, DOI 10.17487/RFC3820, June 2004, <<https://www.rfc-editor.org/rfc/rfc3820>>.
- [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/rfc/rfc5246>>.
- [RFC5912] Hoffman, P. and J. Schaad, "New ASN.1 Modules for the Public Key Infrastructure Using X.509 (PKIX)", RFC 5912, DOI 10.17487/RFC5912, June 2010, <<https://www.rfc-editor.org/rfc/rfc5912>>.

- [RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <<https://www.rfc-editor.org/rfc/rfc8032>>.
- [RFC8555] Barnes, R., Hoffman-Andrews, J., McCarney, D., and J. Kasten, "Automatic Certificate Management Environment (ACME)", RFC 8555, DOI 10.17487/RFC8555, March 2019, <<https://www.rfc-editor.org/rfc/rfc8555>>.
- [ROBOT] Boeck, H., Somorovsky, J., and C. Young, "Return Of Bleichenbacher's Oracle Threat (ROBOT)", 27th USENIX Security Symposium , 2018.
- [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. ASN.1 Module

The following ASN.1 module provides the complete definition of the DelegationUsage certificate extension. The ASN.1 module makes imports from [RFC5912].

```
DelegatedCredentialExtn
{ iso(1) identified-organization(3) dod(6) internet(1)
  security(5) mechanisms(5) pkix(7) id-mod(0)
  id-mod-delegated-credential-extn(95) }

DEFINITIONS IMPLICIT TAGS ::=
BEGIN

-- EXPORT ALL

IMPORTS

EXTENSION
FROM PKIX-CommonTypes-2009 -- From RFC 5912
{ iso(1) identified-organization(3) dod(6) internet(1)
  security(5) mechanisms(5) pkix(7) id-mod(0)
  id-mod-pkixCommon-02(57) } ;

-- OID

id-cloudflare OBJECT IDENTIFIER ::=
{ iso(1) identified-organization(3) dod(6) internet(1) private(4)
  enterprise(1) 44363 }

-- EXTENSION

ext-delegationUsage EXTENSION ::=
{ SYNTAX DelegationUsage
  IDENTIFIED BY id-pe-delegationUsage }

id-pe-delegationUsage OBJECT IDENTIFIER ::= { id-cloudflare 44 }

DelegationUsage ::= NULL

END
```

## Appendix B. Example Certificate

The following certificate has the Delegated Credentials OID.

-----BEGIN CERTIFICATE-----  
MIIFRjCCBMugAwIBAgIQDGeVB+1Y0o/OeCHFSJ6YnTAKBggqhkJOPQQDAzBMMQsw  
CQYDVQQGEwJVUzEVMBMGA1UEChMMRGlNaUNlcnQgSW5jMSYwJAYDVQQDEx1EaWdp  
Q2VydCBFQ0MgU2VjdXJlIFNlcnZlciBDQTAEfW0xOTAzMjYwMDAwMDBaFw0yMTAz  
MzAxMjAwMDBaMGoxCzAJBgNVBAYTA1VTMRMwEQYDVQQIEwppDYWxpZm9ybmlhMRYw  
FAYDVQQHEw1TYW4gRnJhbmNpc2NvMRkwFwYDVQQKEzBDBG91ZGZsYXJlLCBJbmM  
MRMwEQYDVQQDEwppYzJrZG0uY29tMFkwEYHKOZIxj0CAQYIKoZiZj0DAQDCQgAE  
d4az183Bw0FcPgfoeiZpZ2ZnGuxjbgv++wzE0zAj8vniUkKxOWSQiGNLnl+xlWQAL  
lw9djRn1rLmVmn2gb9GgdKOCA28wbgNrMB8GA1UdIwQYMBaAFKOD5h/52j1pWg7o  
kcuVpdox4gqfMB0GA1UdDgQWBBSfcb7fS3fUfAYB91fRcwoDPtgtJjAjbGnVHREE  
HDAaggprYzJrZG0uY29tggwqLmtjMmtkbS5jb20wDgYDVR0PAQH/BAQDAgeAMB0G  
A1UdJQQWMBQGCSGAQUFBwMBBggrBgEFBQcDAjBpBgNVHR8EYjBgMC6gLKAqhiho  
dHRwOi8vY3J5My5kaWdpY2VydC5jb20vc3NjYS1lY2MtZzEuY3J5MC6gLKAqhiho  
dHRwOi8vY3J5NC5kaWdpY2VydC5jb20vc3NjYS1lY2MtZzEuY3J5MEwGA1UdIARF  
MEMwNwYJYIZIAyB9baEBMCCowKAYIKwYBBQUHAQEWHGH0dHBzOi8vd3d3LmRpZ21j  
ZXJ0LmNvbS9DUFMwCAYGZ4EMAQICMHsGCCsGAQUFBwEBBG8wbTakBggrBgEFBQcw  
AYYYaHR0cDovL29jc3AuZGlnaWNlcnQuY29tMEUGCCsGAQUFBzACHjlodHRwOi8v  
Y2FjZjZlcy5kaWdpY2VydC5jb20vRGlnaUNlcnRFQ0NTZWNN1cmVTXjZ2ZXJlZDQ5  
jcnQwDAYDVR0TAQH/BAIwADAPBgkrgEEAYLaSywEAgUAMIIBfGyKKwYBBAHwQIE  
AgSCAw4EggFqAgWAdgC72d+8H4pztZOU15eqkntHOFvECqtS6BqQlmQ2jh7RhQAA  
AWm5hYJ5AAAEAwBHMEUCICiGfq+hStHrL2m8H0awoDR8OpnEHNkF0nI6nL5yYL/j  
AiEAXwebGs/T6Es0YarPzoQrVZqk+sHH/t+jrSrKd5TDjcAdgCHdb/nWXz4jEOZ  
X73zbv9WjUdWnV9KtWDBtOr/XqCDDwAAAWm5hYNgAAAEAwBHMEUCIQD9OWA8KGL6  
bxDKfGileHJWB0iWieRs88VgJyfAg/aFDgIgQ/OsdSF9XOy1foqge0DTDM2FExuw  
0JR0AGZWxOntJzMAAgBELGUus07Or8RAB9io/iJa2uaCvtjLmBU/0zOWtbaBqAAA  
AWm5hYHgAAAEAwBHMEUCIQC4vua1n3BqthEqpA/VBTcsNwMtAwpCuac2IhJ9wx6X  
/Aigb+o00k28JQo9TmPp4vzJ3BD3HXWSNc2Zizbq7mkUQYMWcGyYIKoZiZj0EAwMD  
aQAwZgIXaJsX7d0SuA8ddf/m7Iwfnfs3MQfJyGkEezMJX1t6sRso5z50SS12LpXe  
muGa1FE2ZgIXaL+CDUF5pz7mhrAEIjQ1MqlpF9tH40dJGvYZZQ3W23CmZSkdfvlt  
y5S4RfWHIIPjbw==  
-----END CERTIFICATE-----

## Authors' Addresses

Richard Barnes  
Cisco  
Email: [rlb@ipv.sx](mailto:rlb@ipv.sx)

Subodh Iyengar  
Facebook  
Email: [subodh@fb.com](mailto:subodh@fb.com)

Nick Sullivan  
Cloudflare  
Email: [nick@cloudflare.com](mailto:nick@cloudflare.com)

Eric Rescorla  
Mozilla  
Email: [ekr@rtfm.com](mailto:ekr@rtfm.com)

jhoyle  
Internet-Draft  
Intended status: Standards Track  
Expires: 7 June 2021

J. Hoyland  
Cloudflare Ltd.  
C.A. Wood  
Cloudflare  
4 December 2020

TLS 1.3 Extended Key Schedule  
draft-jhoyle-tls-extended-key-schedule-03

Abstract

TLS 1.3 is sometimes used in situations where it is necessary to inject extra key material into the handshake. This draft aims to describe methods for doing so securely. This key material must be injected in such a way that both parties agree on what is being injected and why, and further, in what order.

Note to Readers

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

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

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 June 2021.

## Copyright Notice

Copyright (c) 2020 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. Conventions and Definitions . . . . .	3
3. Key Schedule Extension . . . . .	3
3.1. Handshake Secret Injection . . . . .	3
3.2. Main Secret Injection . . . . .	3
4. Key Schedule Injection Negotiation . . . . .	4
5. Key Schedule Extension Structure . . . . .	4
6. Security Considerations . . . . .	5
7. IANA Considerations . . . . .	5
8. References . . . . .	5
8.1. Normative References . . . . .	5
8.2. Informative References . . . . .	6
Appendix A. Potential Use Cases . . . . .	6
Acknowledgments . . . . .	7
Authors' Addresses . . . . .	7

## 1. Introduction

Introducing additional key material into the TLS handshake is a non-trivial process because both parties need to agree on the injection content and context. If the two parties do not agree then an attacker may exploit the mismatch in so-called channel synchronization attacks, such as those described by [SLOTH].

Injecting key material into the TLS handshake allows other protocols to be bound to the handshake. For example, it may provide additional protections to the ClientHello message, which in the standard TLS handshake only receives protections after the server's Finished message has been received. It may also permit the use of combined shared secrets, possibly from multiple key exchange algorithms, to be included in the key schedule. This pattern is common for Post Quantum key exchange algorithms, as discussed in

[I-D.ietf-tls-hybrid-design]. In particular, [I-D.ietf-tls-hybrid-design] uses the concatenation pattern described in this draft, but does not add the requisite framing.

The goal of this document is to provide a standardised way for binding extra context into TLS 1.3 handshakes in a way that is easy to analyse from a security perspective, reducing the need for security analysis of extensions that affect the key schedule. It separates the concerns of whether an extension achieves its goals from the concerns of whether an extension reduces the security of a TLS handshake, either directly or through some unforeseen interaction with another extension.

## 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.

## 3. Key Schedule Extension

This section describes two places in which additional secrets can be injected into the TLS 1.3 key schedule.

### 3.1. Handshake Secret Injection

To inject extra key material into the Handshake Secret it is recommended to prefix it, inside an appropriate frame, to the "(EC)DHE" input, where "||" represents concatenation.

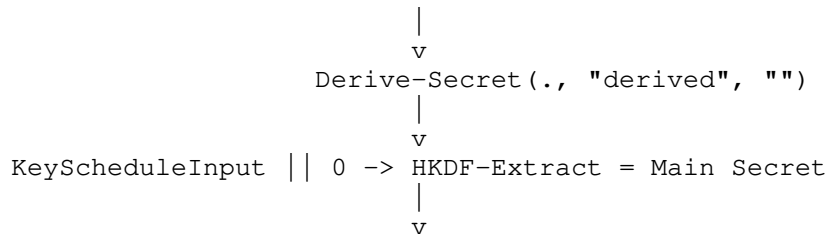
```

      |
      v
Derive-Secret(., "derived", "")
      |
      v
KeyScheduleInput || (EC)DHE -> HKDF-Extract = Handshake Secret
      |
      v

```

### 3.2. Main Secret Injection

To inject key material into the Main Secret it is recommended to prefix it, inside an appropriate frame, to the "0" input.



This structure mirrors the Handshake Injection point.

#### 4. Key Schedule Injection Negotiation

Applications which make use of additional key schedule inputs **MUST** define a mechanism for negotiating the content and type of that input. This input **MUST** be framed in a `KeyScheduleSecret` struct, as defined in Section 5. Applications must take care that any negotiation that takes place unambiguously agrees a secret. It must be impossible, even under adversarial conditions, that a client and server agree on the transcript of the negotiation, but disagree on the secret that was negotiated.

#### 5. Key Schedule Extension Structure

In some cases, protocols may require more than one secret to be injected at a particular stage in the key schedule. Thus, we require a generic and extensible way of doing so. To accomplish this, we use a structure - `KeyScheduleInput` - that encodes well-ordered sequences of secret material to inject into the key schedule. `KeyScheduleInput` is defined as follows:

```

struct {
    KeyScheduleSecretType type;
    opaque secret_data<0..2^16-1>;
} KeyScheduleSecret;

enum {
    (65535)
} KeyScheduleSecretType;

struct {
    KeyScheduleSecret secrets<0..2^16-1>;
} KeyScheduleInput;

```

Each secret included in a `KeyScheduleInput` structure has a type and corresponding secret data. Each secret **MUST** have a unique `KeyScheduleSecretType`. When encoding `KeyScheduleInput` as the key schedule Input value, the `KeyScheduleSecret` values **MUST** be in

ascending sorted order. This ensures that endpoints always encode the same KeyScheduleInput value when using the same secret keying material.

## 6. Security Considerations

[BINDEL] provides a proof that the concatenation approach in Section 3 is secure as long as either the concatenated secret is secure or the existing KDF input is secure.

[[OPEN ISSUE: Is this guarantee sufficient? Do we also need to guarantee that a malicious prefix can't weaken the resulting PRF output?]]

## 7. IANA Considerations

This document requests the creation of a new IANA registry: TLS KeyScheduleInput Types. This registry should be under the existing Transport Layer Security (TLS) Parameters heading. It should be administered under a Specification Required policy [RFC8126].

[[OPEN ISSUE: specify initial registry values]]

Value	Description	DTLS-OK	Reference
TBD	TBD	TBD	TBD

Table 1

## 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>>.
- [RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/info/rfc8126>>.
- [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>>.



## 8.2. Informative References

- [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](https://doi.org/10.1007/978-3-030-25510-7_12)>.
- [I-D.friel-tls-eap-dpp] Friel, O. and D. Harkins, "Bootstrapped TLS Authentication", Work in Progress, Internet-Draft, draft-friel-tls-eap-dpp-01, 13 July 2020, <<http://www.ietf.org/internet-drafts/draft-friel-tls-eap-dpp-01.txt>>.
- [I-D.ietf-tls-hybrid-design] Stebila, D., Fluhrer, S., and S. Gueron, "Hybrid key exchange in TLS 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-hybrid-design-01, 15 October 2020, <<http://www.ietf.org/internet-drafts/draft-ietf-tls-hybrid-design-01.txt>>.
- [I-D.ietf-tls-semistatic-dh] Rescorla, E., Sullivan, N., and C. 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, <<http://www.ietf.org/internet-drafts/draft-ietf-tls-semistatic-dh-01.txt>>.
- [SLOTH] Bhargavan, K. and G. Leurent, "Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH", Proceedings 2016 Network and Distributed System Security Symposium, DOI 10.14722/ndss.2016.23418, 2016, <<https://doi.org/10.14722/ndss.2016.23418>>.

## Appendix A. Potential Use Cases

The draft provides a mechanism for importing additional information into the TLS key schedule. Future applications and specifications can use this mechanism to layer TLS on to other protocols, as opposed to layering other protocols over TLS. For example, as discussed in Section 1, this can be used for hybrid key exchange, which, in effect, is layering TLS over a secondary AKE. Although the key exchanges are interleaved, the post-quantum AKE completes first, as demonstrated by its output key being used as an input for computing TLS's master secret.

This can also be used in more direct ways, such as bootstrapping EAP-TLS as in [I-D.friel-tls-eap-dpp]. This draft also allows for more direct implementations of things such as semi-static DH [I-D.ietf-tls-semistatic-dh]. The aim of this draft is to be sufficiently flexible that it can be used as the basis for layering TLS on top of any protocol that outputs a secure channel binding, where secure is defined by the goals of the overall layered protocol. This draft does not provide security itself, it simply provides a standard format for layering.

#### Acknowledgments

We thank Karthik Bhargavan for his comments.

#### Authors' Addresses

Jonathan Hoyland  
Cloudflare Ltd.

Email: jonathan.hoyland@gmail.com

Christopher A. Wood  
Cloudflare

Email: caw@heapingbits.net

TLS Working Group  
Internet-Draft  
Intended status: Standards Track  
Expires: May 7, 2020

E. Rescorla  
Mozilla  
N. Sullivan  
Cloudflare  
C. Wood  
Apple Inc.  
November 04, 2019

Semi-Static Diffie-Hellman Key Establishment for TLS 1.3  
draft-rescorla-tls-semistatic-dh-02

Abstract

TLS 1.3 [I-D.ietf-tls-tls13] specifies a signed Diffie-Hellman exchange modelled after SIGMA [SIGMA]. This design is suitable for endpoints whose certified credential is a signing key, which is the common situation for current TLS servers. This document describes a mode of TLS 1.3 in which one or both endpoints have a certified DH key which is used to authenticate the exchange.

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 May 7, 2020.

Copyright Notice

Copyright (c) 2019 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. Protocol Overview . . . . .	3
3. Negotiation . . . . .	5
4. Certificate Format . . . . .	5
5. Cryptographic Details . . . . .	5
5.1. Certificate Verify Computation . . . . .	5
5.2. Key Schedule . . . . .	6
6. Client Authentication . . . . .	6
7. Security Considerations . . . . .	6
8. IANA Considerations . . . . .	7
9. References . . . . .	7
9.1. Normative References . . . . .	7
9.2. Informative References . . . . .	8
Authors' Addresses . . . . .	8

## 1. Introduction

DISCLAIMER: This is a work-in-progress draft and has not yet seen significant security analysis. Thus, this draft should not be used as a basis for building production systems.

TLS 1.3 [I-D.ietf-tls-tls13] specifies a signed Diffie-Hellman exchange modeled after SIGMA [SIGMA]. This design is suitable for endpoints whose certified credential is a signing key, which is the common situation for current TLS servers, which is why it was selected for TLS 1.3.

However, it is also possible - although currently rare - for endpoints to have a credential which is an (EC)DH key. This can happen in one of two ways:

- o They may be issued a certificate with an (EC)DH key, as specified for instance in [I-D.ietf-curdle-pkix]
- o They may have a signing key which they use to generate a delegated credential [I-D.ietf-tls-subcerts] containing an (EC)DH key.

In these situations, a signed DH exchange is not appropriate, and instead a design in which the server authenticates via its long-term (EC)DH key is suitable. This document describes such a design modeled on that described in OPTLS [KW16].

This design has a number of potential advantages over the signed exchange in TLS 1.3, specifically:

- o If the end-entity certificate contains an (EC)DH key, TLS can operate with a single asymmetric primitive (Diffie-Hellman). The PKI component will still need signatures, but the TLS stack need not have one. Note that this advantage is somewhat limited if the (EC)DH key is in a delegated credential, but that allows for a clean transition to (EC)DH certificates.
- o It is more resistant to random number generation failures on the server because the attacker needs to have both the server's long-term (EC)DH key and the ephemeral (EC)DH key in order to compute the traffic secrets. [Note: [I-D.irtf-cfrg-randomness-improvements] describes a technique for accomplishing this with a signed exchange.]
- o If the server has a comparatively slow signing cert (e.g., P-256) it can amortize that signature over a large number of connections by creating a delegated credential with an (EC)DH key from a faster group (e.g., X25519).
- o Because there is no signature, the server has deniability for the existence of the communication. Note that it could always have denied the contents of the communication.

This exchange is not generally faster than a signed exchange if comparable groups are used. In fact, if delegated credentials are used, it may be slower on the client as it has to validate the delegated credential, though the result may be cached.

## 2. Protocol Overview

The overall protocol flow remains the same as that in ordinary TLS 1.3, as shown below:



As usual, the client and server each supply an (EC)DH share in their "key\_share" extensions. However, in addition, the server supplies a (signed) static (EC)DH share in its Certificate message, either directly in its end-entity certificate or in a delegated credential. The client and server then perform two (EC)DH exchanges:

- o Between the client and server "key\_share" values to form an ephemeral secret (ES). This is the same value as is computed in TLS 1.3 currently.
- o Between the client's "key\_share" and the server's static share, to form a static secret (SS).

Note that this means that the server's static secret MUST be in the same group as selected group for the ephemeral (EC)DH exchange.

The handshake then proceeds as usual, except that:

- o Instead of containing a signature, the CertificateVerify contains a MAC of the handshake transcript, computed based on SS.
- o SS is mixed into the key schedule at the last HKDF-Extract stage (where currently a 0 is used as the IKM input).

### 3. Negotiation

In order to negotiate this mode, we treat the (EC)DH MAC as if it were a signature and negotiate it with a set of new signature scheme values:

```
enum {  
    sig_p256(0x0901),  
    sig_p384(0x0902),  
    sig_p521(0x0903),  
    sig_x52219(0x0904),  
    sig_x448(0x0905),  
} SignatureScheme;
```

When present in the "signature\_algorithms" extension or CertificateVerify.signature\_scheme, these values indicate DH MAC with the specified key exchange mode. These values MUST NOT appear in "signature\_algorithms\_cert".

Before sending and upon receipt, endpoints MUST ensure that the signature scheme is consistent with the ephemeral (EC)DH group in use.

### 4. Certificate Format

Like signing keys, static DH keys are carried in the Certificate message, either directly in the EE certificate, or in a delegated credential. In either case, the OID for the SubjectPublicKeyInfo MUST be appropriate for use with (EC)DH key establishment. If in a certificate, the key usage and EKU MUST also be set appropriately See [I-D.ietf-curdle-pkix] and [[TBD: P-256, etc.]] for specific details about these formats.

### 5. Cryptographic Details

#### 5.1. Certificate Verify Computation

Instead of a signature, the server proves knowledge of the private key associated with its static share by computing a MAC over the handshake transcript using SS. The transcript thus far includes all messages up to and including Certificate, i.e.:

Transcript-Hash(Handshake Context, Certificate)

The MAC key - SS-Base-Key - is derived from SS as follows:

SS-Base-Key = HKDF-Extract(0, SS)

The MAC is then computed using the Finished computation described in [I-D.ietf-tls-tls13] Section 4.4, with SS-Base-Key as the Base Key value. Receivers MUST validate the MAC and terminate the handshake with a "decrypt\_error" alert upon failure.

Note that this means that the server sends two MAC computations in the handshake, one in CertificateVerify using SS and the other in Finished using the Master Secret. These MACs serve different purposes: the first authenticates the handshake and the second proves possession of the ephemeral secret.

## 5.2. Key Schedule

The final HKDF-Extract stage of the TLS 1.3 key schedule has an HKDF-Extract with the IKM of 0. When static key exchange is negotiated, that 0 is replaced with SS, as shown below.

```

...
    Derive-Secret(., "derived", "")
    |
    v
SS -> HKDF-Extract = Master Secret
    |
    +-----> Derive-Secret(., "c ap traffic",
                        ClientHello...server Finished)
                        = client_application_traffic_secret_0
...

```

## 6. Client Authentication

[[OPEN ISSUE]] In principle, we can do client authentication the same way, with the client's DH key in Certificate and a MAC in CertificateVerify. However, it's less good because the client's static key doesn't get mixed in at all. Also, client DH keys seem even further off.

## 7. Security Considerations

[[OPEN ISSUE: This design requires formal analysis.]]

This is intended to have roughly equivalent security properties to current TLS 1.3, except for the points raised in the introduction.

Open questions:

- o Should semi-static key shares be mixed into the key schedule for client authentication?



- o Should we add support for early data encryption using a semi-static key?

## 8. IANA Considerations

IANA [SHOULD add/has added] the new code points specified in Section 3 to the TLS 1.3 signature scheme registry, with a "recommended" value of TBD.

## 9. References

### 9.1. Normative References

[I-D.ietf-curdle-pkix]

Josefsson, S. and J. Schaad, "Algorithm Identifiers for Ed25519, Ed448, X25519 and X448 for use in the Internet X.509 Public Key Infrastructure", draft-ietf-curdle-pkix-10 (work in progress), May 2018.

[I-D.ietf-httpbis-http2-secondary-certs]

Bishop, M., Sullivan, N., and M. Thomson, "Secondary Certificate Authentication in HTTP/2", draft-ietf-httpbis-http2-secondary-certs-05 (work in progress), November 2019.

[I-D.ietf-tls-exported-authenticator]

Sullivan, N., "Exported Authenticators in TLS", draft-ietf-tls-exported-authenticator-09 (work in progress), May 2019.

[I-D.ietf-tls-subcerts]

Barnes, R., Iyengar, S., Sullivan, N., and E. Rescorla, "Delegated Credentials for TLS", draft-ietf-tls-subcerts-04 (work in progress), July 2019.

[I-D.ietf-tls-tls13]

Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-28 (work in progress), March 2018.

[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>>.

## 9.2. Informative References

- [I-D.irtf-cfrg-randomness-improvements]  
Cremers, C., Garratt, L., Smyshlyaev, S., Sullivan, N.,  
and C. Wood, "Randomness Improvements for Security  
Protocols", draft-irtf-cfrg-randomness-improvements-08  
(work in progress), November 2019.
- [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>>.
- [SIGMA] Krawczyk, H., "SIGMA: the 'SIGn-and-MAC' approach to  
authenticated Diffie-Hellman and its use in the IKE  
protocols", Proceedings of CRYPTO 2003 , 2003.

## Authors' Addresses

Eric Rescorla  
Mozilla  
  
Email: [ekr@rtfm.com](mailto:ekr@rtfm.com)

Nick Sullivan  
Cloudflare  
  
Email: [nick@cloudflare.com](mailto:nick@cloudflare.com)

Christopher A. Wood  
Apple Inc.  
One Apple Park Way  
Cupertino, California 95014  
United States of America  
  
Email: [cawood@apple.com](mailto:cawood@apple.com)