RFC 9261
Exported Authenticators in TLS

Abstract

This document describes a mechanism that builds on Transport Layer Security (TLS) or Datagram Transport Layer Security (DTLS) and enables peers to provide 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 is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9261.

Copyright Notice

Table of Contents

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 that do not have a single certificate that includes all of the identities, can authenticate additional identities over a single connection.

spontaneous authentication: After a connection is established, endpoints can authenticate in response to events in a higher-layer protocol; they can also integrate 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 TLS 1.3 Section 4.6.2 of [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 [RFC9113] 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 [RFC5246] and DTLS 1.2 [RFC6347]. (These were obsoleted by TLS 1.3 [RFC8446] and DTLS 1.3 [RFC9147].)

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, and 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", that 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 that identifies the authenticator request and that 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 precomputing valid authenticators. For example, the application may choose this value to correspond to a value used in an existing data structure in the software to simplify implementation.

extensions: The set of extensions allowed in the structures of CertificateRequest and ClientCertificateRequest is comprised of those defined in the "TLS ExtensionType Values" IANA registry containing CR in the "TLS 1.3" column (see [IANA-TLS] and [RFC8447]). In addition, the set of extensions in the ClientCertificateRequest structure **MAY** include the server_name extension [RFC6066].

The uniqueness requirements of the certificate_request_context apply across CertificateRequest and ClientCertificateRequest messages that are used as part of authenticator requests. 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 (Message Authentication Code) 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 because 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 ciphersuite (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 ciphersuites that do not use the TLS PRF and with TLS 1.3 ciphersuites 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 Certificate, CertificateVerify, and Finished messages [RFC8446]. 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 Section 4.4.2 of [RFC8446].

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; however, it **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" extensions, as described in Section 4.2.3 of [RFC8446] for (D)TLS 1.3 or in 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 Raw Public Keys as described in [RFC7250] are not supported in this version of the specification and their use in this context has not yet been analyzed.

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 Elliptic Curve Digital Signature Algorithm (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 that serves as the separator, and
  • 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 5.1. 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 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 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)
- the 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] (obsoleted by [RFC9162]), etc.)
- a signer (either the private key associated with the identity or the 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 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 be implemented in the TLS library. Implementing this in the TLS library lets the implementer take advantage of existing extension and certificate selection logic, and the implementer can 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 or not the authenticator is valid 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 has updated the entry for server_name(0) in the "TLS ExtensionType Values" registry [IANA-TLS] (defined in [RFC8446]) by replacing the value in the "TLS 1.3" column with the value "CH, EE, CR" and listing this document in the "Reference" column.

IANA has also added 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 has added the following entries to the "TLS Exporter Labels" registry [IANA-EXPORT] (defined in [RFC5705]): "EXPORTER-client authenticator handshake context", "EXPORTER-server 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 listed as the reference.

8.3. Update of the TLS HandshakeType Registry

IANA has added the following entry to the "TLS HandshakeType" registry [IANA-HANDSHAKE] (defined in [RFC8446]): "client_certificate_request" (17) with "DTLS-OK" set to "Y" and this document listed as the reference. In addition, the following appears in the "Comment" 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 that now has been analyzed several times. For example, [SIGMAC] presents a relevant framework for analysis, and Appendix E.1.6 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"; therefore, they cannot be transplanted into other protocols.

In TLS 1.3, the client cannot 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 message. To avoid disagreement between the client and server on the authentication status of Exported Authenticators, servers **MUST** verify the client Finished message before sending an EA or processing a received Exported Authenticator.

10. References

10.1. 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>.

[RFC9147]   Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", RFC 9147, DOI 10.17487/RFC9147, April 2022, <https://www.rfc-editor.org/info/rfc9147>.

10.2. Informative References

[IANA-EXPORT]   IANA, "TLS Exporter Labels", <https://www.iana.org/assignments/tls-parameters/>.

[IANA-HANDSHAKE]   IANA, "TLS HandshakeType", <https://www.iana.org/assignments/tls-parameters/>.

[IANA-TLS]   IANA, "TLS ExtensionType Values", <https://www.iana.org/assignments/tls-extensiontype-values/>.

[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>.

[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>.

[RFC9113]   Thomson, M., Ed. and C. Benfield, Ed., "HTTP/2", RFC 9113, DOI 10.17487/RFC9113, June 2022, <https://www.rfc-editor.org/info/rfc9113>.

[RFC9162]   Laurie, B., Messeri, E., and R. Stradling, "Certificate Transparency Version 2.0", RFC 9162, DOI 10.17487/RFC9162, December 2021, <https://www.rfc-editor.org/info/rfc9162>.

[SIGMAC]   Krawczyk, H., "A Unilateral-to-Mutual Authentication Compiler for Key Exchange (with Applications to Client Authentication in TLS 1.3)", Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, DOI 10.1145/2976749.2978325, August 2016, <https://eprint.iacr.org/2016/711.pdf>.

Acknowledgements

Author's Address

Nick Sullivan
Cloudflare Inc.
Email: nick@cloudflare.com