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

N. Sullivan
Cloudflare Inc.

November 04, 2019

# Exported Authenticators in TLS
# draft-ietf-tls-exported-authenticator-10

Abstract

This document describes a mechanism in Transport Layer Security (TLS) for peers to provide a proof of ownership of a certificate. This proof can be exported by one peer, transmitted out-of-band to the other peer, and verified by the receiving peer.

Table of Contents

1. Introduction . . . . . 2
2. Conventions and Terminology . . . . . 3
3. Message Sequences . . . . . 3
4. Authenticator Request . . . . . 4
5. Authenticator . . . . . 5
5.1. Authenticator Keys . . . . . 5
5.2. Authenticator Construction . . . . . 6
5.2.1. Certificate . . . . . 6
5.2.2. CertificateVerify . . . . . 7
5.2.3. Finished . . . . . 8
5.2.4. Authenticator Creation . . . . . 9
6. Empty Authenticator . . . . . 9
7. API considerations . . . . . 9
7.1. The "request" API . . . . . 10
7.2. The "get context" API . . . . . 10
7.3. The "authenticate" API . . . . . 10
7.4. The "validate" API . . . . . 11
8. IANA Considerations . . . . . 11
8.1. Update of the TLS ExtensionType Registry . . . . . 11
8.2. Update of the TLS Exporter Labels Registry . . . . . 11
9. Security Considerations . . . . . 11
10. Acknowledgements . . . . . 12
11. References . . . . . 12
11.1. Normative References . . . . . 12
11.2. Informative References . . . . . 13
Author's Address . . . . . 13

## 1. Introduction

This document provides a way to authenticate one party of a Transport Layer Security (TLS) connection to its peer using a certificate 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 natively provides:

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.

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, 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 at the TLS layer.

TLS (or DTLS) version 1.2 or later are REQUIRED to implement the mechanisms described in this document.

## 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.

## 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 TLS connection using data exported from that connection. It can be transmitted to the other party of the TLS connection at the application layer. The application layer protocol used to send the authenticator request SHOULD use TLS as its underlying transport to keep the request confidential. The application MAY use the existing TLS connection to transport the authenticator.

An authenticator request message can be constructed by either the client or the server. This authenticator request uses the CertificateRequest message structure from Section 4.3.2 of [TLS13], even if the TLS connection protocol is TLS 1.2. The CertificateRequest is used to define the parameters in a request for an authenticator. This message does not include any TLS framing and is not encrypted with a handshake key.

The uniqueness requirements of the certificate_request_context apply only to CertificateRequest messages that are used as part of authenticator requests. 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. The structure is defined to be:

```
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 certificate 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.

extensions: The extensions that are allowed in this structure include the extensions defined for CertificateRequest messages defined in Section 4.2. of [TLS13] and the server_name [RFC6066] extension, which is allowed for client-generated authenticator requests.

## 5. Authenticator

The authenticator is a structured message that can be exported from either party of a TLS connection. It can be transmitted to the other party of the TLS connection at the application layer. The application layer protocol used to send the authenticator SHOULD use TLS as its underlying transport to keep the certificate confidential. The application MAY use the existing TLS connection to transport the authenticator.

An authenticator message can be constructed by either the client or the server given an established TLS connection, a 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 TLS connection. These values are derived using an exporter as described in [RFC5705] (for TLS 1.2) or Sec. 7.5 of [TLS13] (for TLS 1.3). For 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 and 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 and 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 selected in TLS for the pseudorandom function (PRF). Exported authenticators cannot be used with cipher suites that do not use the TLS PRF and have not defined a hash function for this purpose. This hash is referred to as the authenticator hash.

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

### 5.2. Authenticator Construction

An authenticator is formed from the concatenation of TLS 1.3 [TLS13] Certificate, CertificateVerify, and Finished messages.

If the peer creating the certificate_request_context 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 TLS handshake's ClientHello. Only servers can provide an authenticator without a corresponding request.

ClientHello extensions are used to determine permissible extensions in the Certificate message. This follows the general model for extensions in 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, to ensure that the recipient will be able to process such extensions.

#### 5.2.1. Certificate

The Certificate message contains the certificate to be used for authentication and any supporting certificates in the chain. This structure is defined in [TLS13], 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.

The certificates chosen in the Certificate message MUST conform to the requirements of a Certificate message in the negotiated version of TLS. In particular, the certificate chain MUST be valid for the a signature algorithms indicated by the peer in the "signature_algorithms" and "signature_algorithms_cert" extension, as described in Section 4.2.3 of [TLS13] for TLS 1.3 or the "signature_algorithms" extension from Sections 7.4.2 and 7.4.6 of [RFC5246] for TLS 1.2.

In addition to "signature_algorithms" and "signature_algorithms_cert", the "server_name" [RFC6066], "certificate_authorities" (Section 4.2.4. of [TLS13]), and "oid_filters" (Section 4.2.5. of [TLS13]) extensions are used to guide certificate selection.

Only the X509 certificate type defined in [TLS13] 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 TLS handshake. 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 certificate. The definition for TLS 1.3 is:

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

The algorithm field specifies the signature algorithm used (see Section 4.2.3 of [TLS13] 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 authenticator request. Otherwise, the signature algorithm used should be chosen from the "signature_algorithms" sent by the peer in the ClientHello of the 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 NULL-terminated)
- A single 0 byte 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 (that is, 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 before taking a user-visible action.

#### 5.2.3. Finished

A HMAC [HMAC] over the hashed authenticator transcript, which is the concatenated 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, a constant-time comparison SHOULD be used.

## 6. Empty Authenticator

If, given an authenticator request, the endpoint does not have an appropriate certificate 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 does not include any TLS framing.

```
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 TLS library even if it is possible to implement it at the application layer. 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 TLS version of 1.1 or earlier, or
- the connection is TLS 1.2 and the extended master secret extension [RFC7627] was not negotiated

The following sections describes 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 bytes)
- set of extensions to include (this MUST include signature_algorithms)

It returns an authenticator request, which is a sequence of octets that comprises a CertificateRequest 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 an active connection
- a set of certificate chains and associated extensions (OCSP, SCT, etc.)
- a signer (either the private key associated with the certificate, or interface to perform private key operations) for each chain
- an authenticator request or certificate_request_context (from 0 to 255 bytes)

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 an active connection
- an optional authenticator request
- an authenticator

It returns the certificate chain and extensions and a status to indicate whether the authenticator is valid or not. If the authenticator was empty - that is, it did not contain a certificate - the certificate chain will contain no certificates. The API SHOULD return a failure if the certificate_request_context of the authenticator was used in a previously validated authenticator. Well-formed empty authenticators are returned as valid.

## 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 [TLS13]) by replacing the value in the "TLS 1.3" column with the value "CH, EE, CR".

### 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-server authenticator handshake context", "EXPORTER-client authenticator finished key" and "EXPORTER-server authenticator finished key".

## 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.

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 certificates, rather than individually authoritative over each.

- There is no indication in the TLS layer 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.

## 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>.

[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>.

[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>.

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

### 11.2. Informative References

[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