Workgroup: TLS
Internet-Draft:
draft-fossati-tls-attestation-05
Published: 4 March 2024
Intended Status: Standards Track
Expires: 5 September 2024
Authors: H. Tschofenig, Y. Sheffer (Intuit), P. Howard (Arm Limited), I. Mihalcea (Arm Limited), Y. Deshpande (Arm Limited), A. Niemi (Huawei)

Using Attestation in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)

Abstract

The TLS handshake protocol allows authentication of one or both peers using static, long-term credentials. In some cases, it is also desirable to ensure that the peer runtime environment is in a secure state. Such an assurance can be achieved using attestation which is a process by which an entity produces evidence about itself that another party can use to appraise whether that entity is found in a secure state. This document describes a series of protocol extensions to the TLS 1.3 handshake that enables the binding of the TLS authentication key to a remote attestation session. This enables an entity capable of producing attestation evidence, such as a confidential workload running in a Trusted Execution Environment (TEE), or an IoT device that is trying to authenticate itself to a network access point, to present a more comprehensive set of security metrics to its peer. These extensions have been designed to allow the peers to use any attestation technology, in any remote attestation topology, and mutually.

About This Document

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

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-fossati-tls-attestation/.

Source for this draft and an issue tracker can be found at https://github.com/yaronf/draft-tls-attestation.

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 2024.

Copyright Notice

Copyright (c) 2024 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
3. Overview
4. Attestation Extensions
5. Use of Remote Attestation Credentials in the TLS Handshake
  5.1. Handshake Overview
  5.2. TLS Client Authenticating Using Evidence
  5.3. TLS Server Authenticating Using Evidence
  5.4. TLS Client Authenticating Using Attestation Results
  5.5. TLS Server Authenticating Using Results
6. Evidence Extensions (Background Check Model)
  6.1. Attestation-only
  6.2. Attestation Alongside X.509 Certificates
7. Attestation Results Extensions (Passport Model)
8. TLS Client and Server Handshake Behavior
  8.1. Background Check Model
    8.1.1. Client Hello
    8.1.2. Server Hello
  8.2. Passport Model
    8.2.1. Client Hello
    8.2.2. Server Hello

9. Background-Check Model Examples
  9.1. Cloud Confidential Computing
  9.2. IoT Device Onboarding
10. Security Considerations
11. IANA Considerations
  11.1. TLS Extensions
  11.2. TLS Alerts
  11.3. TLS Certificate Types
12. References
  12.1. Normative References
  12.2. Informative References
Appendix A. Design Rationale: X.509 and Attestation Usage Variants
Appendix B. Cross-protocol Binding Mechanism
  B.1. Binding Mechanism
  B.2. Usage
Appendix C. History
  C.1. draft-fossati-tls-attestation-02
  C.2. draft-fossati-tls-attestation-01
  C.3. draft-fossati-tls-attestation-00
Appendix D. Working Group Information
Authors' Addresses

1. Introduction

Attestation [RFC9334] is the process by which an entity produces evidence about itself that another party can use to evaluate the trustworthiness of that entity. This document describes a series of protocol extensions to the TLS 1.3 handshake that enables the binding of the TLS authentication key to a remote attestation session. As a result, a peer can use "attestation credentials", consisting of compound platform evidence and key attestation, to authenticate itself to its peer during the setup of the TLS channel. This enables an attester, such as a confidential workload running in a Trusted Execution Environment (TEE) [I-D.ietf-teep-architecture], or an IoT device that is trying to authenticate itself to a network access point, to present a more comprehensive set of security metrics to its peer. This, in turn, allows for the implementation of authorization policies at the relying parties that are based on stronger security signals.

Given the variety of deployed and emerging attestation technologies (e.g., [TPM1.2], [TPM2.0], [I-D.ietf-rats-eat]) these extensions have been explicitly designed to be agnostic of the attestation formats. This is achieved by reusing the generic encapsulation defined in [I-D.ietf-rats-msg-wrap] for transporting evidence and attestation result payloads in the TLS Certificate message.

The proposed design supports both background-check and passport topologies, as described in Sections 5.2 and 5.1 of [RFC9334]. This

is detailed in Section 6 and Section 7. This specification provides both one-way (server-only) and mutual (client and server) authentication using attestation credentials, and allows the attestation topologies at each peer to be independent of each other.

This document does not specify any attestation technology. Companion documents are expected to define specific attestation mechanisms.

2. Conventions and Terminology

The reader is assumed to be familiar with the vocabulary and concepts defined in Section 4 of [RFC9334], and those in Section 2 of [I-D.bft-rats-kat].

The following terms are used in this document:

TLS Identity Key (TIK):
A cryptographic key used by one of the peers to authenticate itself during the TLS handshake.

TIK-C, TIK-S:
The TIK that identifies the client or the server, respectively.

TIK-C-ID, TIK-S-ID:
An identifier for TIK-C or respectively, TIK-S. This may be a fingerprint (cryptographic hash) of the public key, but other implementations are possible.

"Remote attestation credentials", or "attestation credentials", is used to refer to both attestation evidence and attestation results, when no distinction needs to be made between them.

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. Overview

The basic functional goal is to link the authenticated key exchange of TLS with an interleaved remote attestation session in such a way that the key used to sign the handshake can be proven to be residing within the boundaries of an attested TEE. The requirement is that the attester can provide evidence containing the security status of both the signing key and the platform that is hosting it. The associated security goal is to obtain such binding so that no replay, relay or splicing from an adversary is possible.

Throughout the document, we will assume the conceptual attester model described in Section 3 of [I-D.bft-rats-kat], where TEE attestation is provided by a Platform Attestation Token (PAT) signed by the attester's "attesting environment". Among other security metrics, the PAT contains evidence about the integrity of a "Key Attestation Service" executing within the TEE which issues a Key Attestation Token (KAT) for the TLS handshake signing key (TIK) as described in Section 5.1.

The protocol's security relies on the verifiable binding between these two logically separate units of evidence.

4. Attestation Extensions

As typical with new features in TLS, the client indicates support for the new extension in the ClientHello message. The newly introduced extensions allow remote attestation credentials and nonces to be exchanged. The nonces are used for guaranteeing freshness of the exchanged evidence when the background check model is in use.

When either the evidence or the attestation results extension is successfully negotiated, the content of the corresponding Certificate message contains a payload that is encoded based on the wrapper defined in [I-D.ietf-rats-msg-wrap]. Both JSON and CBOR serializations are allowed in CMW, with the emitter choosing which serialization to use.

In TLS a client has to demonstrate possession of the private key via the CertificateVerify message, when client-based authentication is requested. The attestation payload must contain assertions relating to the client's TLS Identity Key (TIK-C), which associate the private key with the attestation information. These assertions may come in the form of a Key Attestation Token (KAT), or of specific claims in an attestation result document. An example of a KAT format utilizing the EAT format can be found in [I-D.bft-rats-kat].

The relying party can obtain and appraise the remote attestation results either directly from the Certificate message (in the passport model), or by relaying the evidence from the Certificate message to the verifier and receiving the attestation results. Subsequently, the attested key is used to verify the CertificateVerify message.

When using the passport model, the remote attestation results obtained by the attester from its trusted verifiers can be cached and used for any number of subsequent TLS handshakes, as long as the freshness policy requirements are satisfied.

This protocol supports both monolithic and split implementations. In a monolithic implementation, the TLS stack is completely embedded within the TEE. In a split implementation, the TLS stack is located outside the TEE, but any private keys (and in particular, the TIK) only exist within the TEE. In order to support both options, only the TIK's identity and its public component are ever passed between the Client or Server TLS stack and its Attestation Service.

5. Use of Remote Attestation Credentials in the TLS Handshake

For both the passport model (described in section 5.1 of [RFC9334]) and background check model (described in Section 5.2 of [RFC9334]) the following modes of operation are allowed when used with TLS, namely:

* TLS client is the attester,
* TLS server is the attester, and
* TLS client and server mutually attest towards each other.

We will show the message exchanges of the first two cases in sub-sections below. Mutual authentication via attestation combines these two (non-interfering) flows, including cases where one of the peers uses the passport model for its attestation, and the other uses the background check model.

5.1. Handshake Overview

The handshake defined here is analogous to certificate-based authentication in a regular TLS handshake. Instead of the certificate's private key, we use the TIK identity key. This key is attested, with attestation being carried by the Certificate message. Following that, the peer being attested proves possession of the private key using the CertificateVerify message.

Depending on the use case, the protocol supports peer authentication using attestation only, or using both attestation and a regular public key certificate.

The current version of the document assumes the KAT/PAT construct of [I-D.bft-rats-kat]. Not all platforms support this model, and a document that defines private key attestation for use in TLS Attestation as defined here, must specify:

* The format and the lifetime of TIK (e.g. an ephemeral, per session TIK vs. a long lived one).
* How the key is attested using a structure carried by the Certificate message.

*How proof of possession is performed.

5.2. TLS Client Authenticating Using Evidence

In this use case, the TLS server (acting as a relying party) challenges the TLS client (as the attester) to provide evidence. The TLS server needs to provide a nonce in the EncryptedExtensions message to the TLS client so that the attestation service can feed the nonce into the generation of the evidence. The TLS server, when receiving the evidence, will have to contact the verifier (which is not shown in the diagram).

An example of this flow can be found in device onboarding where the client initiates the communication with cloud infrastructure to get credentials, firmware and other configuration data provisioned to the device. For the server to consider the device genuine it needs to present evidence.

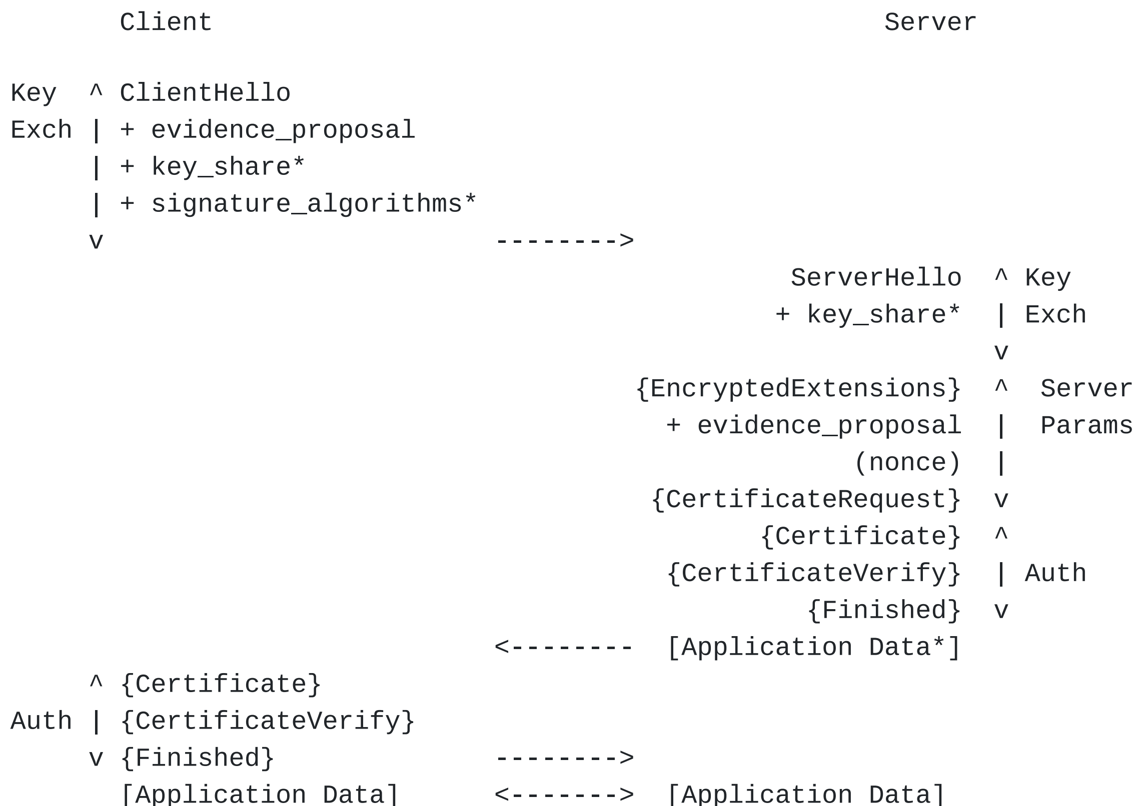


Figure 1: TLS Client Providing Evidence to TLS Server.

5.3. TLS Server Authenticating Using Evidence

In this use case the TLS client challenges the TLS server to present evidence. The TLS server acts as an attester while the TLS client is the relying party. The TLS client, when receiving the evidence, will have to contact the verifier (which is not shown in the diagram).

An example of this flow can be found in confidential computing where a compute workload is only submitted to the server infrastructure once the client/user is assured that the confidential computing platform is genuine.

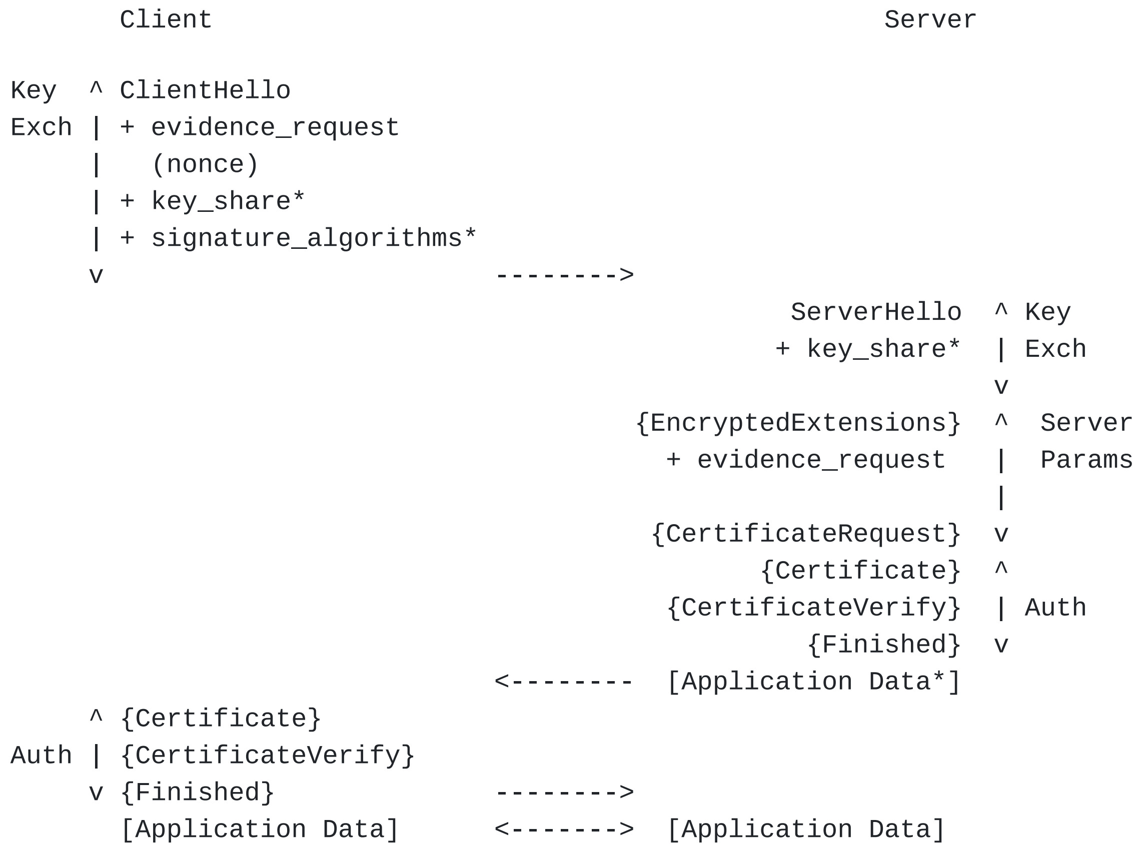


Figure 2: TLS Server Providing Evidence to TLS Client.

5.4. TLS Client Authenticating Using Attestation Results

In this use case the TLS client, as the attester, provides attestation results to the TLS server. The TLS client is the attester and the the TLS server acts as a relying party. Prior to delivering its Certificate message, the client must contact the verifier (not shown in the diagram) to receive the attestation results that it will use as credentials.

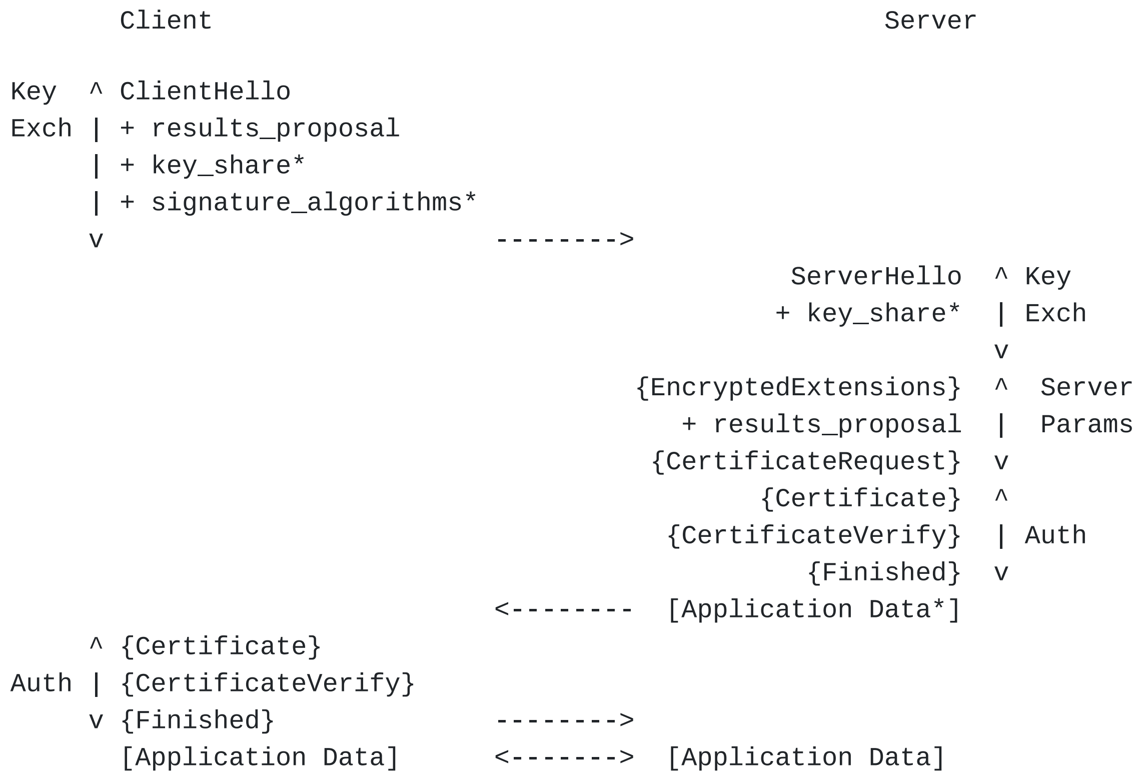


Figure 3: TLS Client Providing Results to TLS Server.

5.5. TLS Server Authenticating Using Results

In this use case the TLS client, as the relying party, requests attestation results from the TLS server. Prior to delivering its Certificate message, the server must contact the verifier (not shown in the diagram) to receive the attestation results that it will use as credentials.

Figure 4: TLS Server Providing Attestation Results to TLS Client.

6. Evidence Extensions (Background Check Model)

The EvidenceType structure also contains an indicator for the type of credential expected in the Certificate message. The credential can either contain attestation evidence alone, or an X.509 certificate alongside attestation evidence.

```
enum { CONTENT_FORMAT(0), MEDIA_TYPE(1) } typeEncoding;
enum { ATTESTATION(0), CERT_ATTESTATION(1) } credentialKind;

struct {
    credentialKind credential_kind;
    typeEncoding type_encoding;
    select (EvidenceType.type_encoding) {
        case CONTENT_FORMAT:
            uint16 content_format;
        case MEDIA_TYPE:
            opaque media_type<0..2^16-1>;
    };
} EvidenceType;

struct {
    select(Handshake.msg_type) {
        case client_hello:
            EvidenceType supported_evidence_types<1..2^8-1>;
            opaque nonce<8..2^8-1>;
        case server_hello:
            EvidenceType selected_evidence_type;
    }
} evidenceRequestTypeExtension;

struct {
    select(Handshake.msg_type) {
        case client_hello:
            EvidenceType supported_evidence_types<1..2^8-1>;
        case server_hello:
            EvidenceType selected_evidence_type;
            opaque nonce<8..2^8-1>;
    }
} evidenceProposalTypeExtension;
```

Figure 5: TLS Extension Structure for Evidence.

6.1. Attestation-only

When the chosen evidence type indicates the sole use of attestation for authentication, the Certificate payload is used as a container for attestation evidence, as shown in Figure 6, and follows the model of [RFC8446].

```
struct {
    select (certificate_type) {
        case RawPublicKey:
            /* From RFC 7250 ASN.1_subjectPublicKeyInfo */
            opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;

          case X509:
            opaque cert_data<1..2^24-1>;

          case attestation:
              /* payload used to convey evidence */
              opaque evidence<1..2^24-1>;
      };
      Extension extensions<0..2^16-1>;
  } CertificateEntry;

  struct {
      opaque certificate_request_context<0..2^8-1>;
      CertificateEntry certificate_list<0..2^24-1>;
  } Certificate;
```

Figure 6: Certificate Message when using only attestation.

The encoding of the evidence structure is defined in [I-D.ietf-rats-msg-wrap].

6.2. Attestation Alongside X.509 Certificates

When the chosen evidence type indicates usage of both attestation and PKIX, the X.509 certificate will serve as the main payload in the Certificate message, while the attestation evidence will be carried in the CertificateEntry extension, as shown in Figure 7.

```
struct {
    select (certificate_type) {
        case RawPublicKey:
            /* From RFC 7250 ASN.1_subjectPublicKeyInfo */
            opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;

        case X509:
            /* X.509 certificate conveyed as usual */
            opaque cert_data<1..2^24-1>;
    };

    /* attestation evidence conveyed as an extension, see below */
    Extension extensions<0..2^16-1>;
  } CertificateEntry;

struct {
    opaque certificate_request_context<0..2^8-1>;
    CertificateEntry certificate_list<0..2^24-1>;
} Certificate;

struct {
    ExtensionType extension_type;
    /* payload used to convey evidence */
    opaque extension_data<0..2^16-1>;
} Extension;

enum {
    /* other extension types defined in the IANA TLS
       ExtensionType Value registry */

    /* variant used to identify attestation evidence */
    attestation_evidence(60),
    (65535)
} ExtensionType;
```

Figure 7: Certificate Message when using PKIX and attestation.

The encoding of the evidence structure is defined in [I-D.ietf-rats-msg-wrap].

As described in Appendix A, this authentication mechanism is meant primarily for carrying platform attestation evidence to provide more context to the relying party. This evidence must be cryptographically bound to the TLS handshake to prevent relay attacks. An Attestation Channel Binder as described in Appendix B is therefore used when the attestation scheme does not allow the binding data to be part of the token. The structure of the binder is given in Figure 8.

```
attestation_channel_binder = {
  &(nonce: 1) => bstr .size (8..64)
  &(ik_pub_fingerprint: 2) => bstr .size (16..64)
  &(channel_binder: 3) => bstr .size (16..64)
}
```

Figure 8: Format of TLS channel binder.

* Nonce is the value provided as a challenge by the relying party.

* The identity key fingerprint (ik_pub_fingerprint) is a hash of the Subject Public Key Info from the leaf X.509 certificate transmitted in the handshake.

* The channel binder (channel_binder) is a value obtained from the early exporter mechanism offered by the TLS implementation (Section 7.5 of [RFC8446]). This Early Exporter Value (EEV) must be obtained immediately following the ServerHello message, using 'attestation-binder' as the label, an empty context, and with the key length set to 32 bytes. Figure 9 shows this computation using the notation from [RFC8446].

```
TLS-Early-Exporter(label, context_value, key_length) =
       HKDF-Expand-Label(
              Derive-Secret(early_exporter_master_secret, label, ""),
              "exporter", Hash(context_value), key_length)

channel_binder = TLS-Early-Exporter(label = "attestation-binder",
                                    context_value = "", key_length = 32)
```

Figure 9: Usage of TLS v1.3 early exporter for channel binding.

A hash of the binder must be included in the attestation evidence. Previous to hashing, the binder must be encoded as described in Appendix B.

The hash algorithm negotiatied within the handshake must be used wherever hashing is required for the binder.

7. Attestation Results Extensions (Passport Model)

```
struct {
    opaque verifier_identity<0..2^16-1>;
} VerifierIdentityType;

struct {
    select(Handshake.msg_type) {
        case client_hello:
            VerifierIdentityType trusted_verifiers<1..2^8-1>;

        case server_hello:
            VerifierIdentityType selected_verifier;
    }
} resultsRequestTypeExtension;

struct {
    select(Handshake.msg_type) {
        case client_hello:
            VerifierIdentityType trusted_verifiers<1..2^8-1>;

        case server_hello:
            VerifierIdentityType selected_verifier;
    }
} resultsProposalTypeExtension;
```

Figure 10: TLS Extension Structure for Attestation Results.

8. TLS Client and Server Handshake Behavior

The high-level message exchange in Figure 11 shows the evidence_proposal, evidence_request, results_proposal, and results_request extensions added to the ClientHello and the EncryptedExtensions messages.

Figure 11: Attestation Message Overview.

8.1. Background Check Model

8.1.1. Client Hello

To indicate the support for passing evidence in TLS following the background check model, clients include the evidence_proposal and/or the evidence_request extensions in the ClientHello.

The evidence_proposal extension in the ClientHello message indicates the evidence types the client is able to provide to the server, when requested using a CertificateRequest message.

The evidence_request extension in the ClientHello message indicates the evidence types the client challenges the server to provide in a subsequent Certificate payload.

The evidence_proposal and evidence_request extensions sent in the ClientHello each carry a list of supported evidence types, sorted by

preference. When the client supports only one evidence type, it is a list containing a single element.

The client **MUST** omit evidence types from the evidence_proposal extension in the ClientHello if it cannot respond to a request from the server to present a proposed evidence type, or if the client is not configured to use the proposed evidence type with the given server. If the client has no evidence types to send in the ClientHello it **MUST** omit the evidence_proposal extension in the ClientHello.

The client **MUST** omit evidence types from the evidence_request extension in the ClientHello if it is not able to pass the indicated verification type to a verifier. If the client does not act as a relying party with regards to evidence processing (as defined in the RATS architecture) then the client **MUST** omit the evidence_request extension from the ClientHello.

8.1.2. Server Hello

If the server receives a ClientHello that contains the evidence_proposal extension and/or the evidence_request extension, then three outcomes are possible:

* The server does not support the extensions defined in this document. In this case, the server returns the EncryptedExtensions without the extensions defined in this document.

* The server supports the extensions defined in this document, but it does not have any evidence type in common with the client. Then, the server terminates the session with a fatal alert of type "unsupported_evidence".

* The server supports the extensions defined in this document and has at least one evidence type in common with the client. In this case, the processing rules described below are followed.

The evidence_proposal extension in the ClientHello indicates the evidence types the client is able to provide to the server, when challenged using a certificate_request message. If the server wants to request evidence from the client, it **MUST** include the evidence_proposal extension in the EncryptedExtensions. This evidence_proposal extension in the EncryptedExtensions then indicates what evidence format the client is requested to provide in a subsequent Certificate message. The value conveyed in the evidence_proposal extension by the server **MUST** be selected from one of the values provided in the evidence_proposal extension sent in the ClientHello. The server **MUST** also send a certificate_request message.

If the server does not send a certificate_request message or none of the evidence types supported by the client (as indicated in the evidence_proposal extension in the ClientHello) match the server-supported evidence types, then the evidence_proposal extension in the ServerHello **MUST** be omitted.

The evidence_request extension in the ClientHello indicates what types of evidence the client can challenge the server to return in a subsequent Certificate message. With the evidence_request extension in the EncryptedExtensions, the server indicates the evidence type carried in the Certificate message sent by the server. The evidence type in the evidence_request extension **MUST** contain a single value selected from the evidence_request extension in the ClientHello.

8.2. Passport Model

8.2.1. Client Hello

To indicate the support for passing attestation results in TLS following the passport model, clients include the results_proposal and/or the results_request extensions in the ClientHello message.

The results_proposal extension in the ClientHello message indicates the verifier identities from which it can relay attestation results, when requested using a CertificateRequest message.

The results_request extension in the ClientHello message indicates the verifier identities from which the client expects the server to provide attestation results in a subsequent Certificate payload.

The results_proposal and results_request extensions sent in the ClientHello each carry a list of supported verifier identities, sorted by preference. When the client supports only one verifier, it is a list containing a single element.

The client **MUST** omit verifier identities from the results_proposal extension in the ClientHello if it cannot respond to a request from the server to present attestation results from a proposed verifier, or if the client is not configured to relay the results from the proposed verifier with the given server. If the client has no verifier identities to send in the ClientHello it **MUST** omit the results_proposal extension in the ClientHello.

The client **MUST** omit verifier identities from the results_request extension in the ClientHello if it is not configured to trust attestation results issued by said verifiers. If the client does not act as a relying party with regards to the processing of attestation results (as defined in the RATS architecture) then the client **MUST** omit the results_request extension from the ClientHello.

8.2.2. Server Hello

If the server receives a ClientHello that contains the results_proposal extension and/or the results_request extension, then three outcomes are possible:

* The server does not support the extensions defined in this document. In this case, the server returns the EncryptedExtensions without the extensions defined in this document.

* The server supports the extensions defined in this document, but it does not have any trusted verifiers in common with the client. Then, the server terminates the session with a fatal alert of type "unsupported_verifiers".

* The server supports the extensions defined in this document and has at least one trusted verifier in common with the client. In this case, the processing rules described below are followed.

The results_proposal extension in the ClientHello indicates the verifier identities from which the client is able to provide attestation results to the server, when challenged using a certificate_request message. If the server wants to request evidence from the client, it **MUST** include the results_proposal extension in the EncryptedExtensions. This results_proposal extension in the EncryptedExtensions then indicates what verifier the client is requested to provide attestation results from in a subsequent Certificate message. The value conveyed in the results_proposal extension by the server **MUST** be selected from one of the values provided in the results_proposal extension sent in the ClientHello. The server **MUST** also send a certificate_request message.

If the server does not send a certificate_request message or none of the verifier identities proposed by the client (as indicated in the results_proposal extension in the ClientHello) match the server-trusted verifiers, then the results_proposal extension in the ServerHello **MUST** be omitted.

The results_request extension in the ClientHello indicates what verifiers the client trusts as issuers of attestation results for the server. With the results_request extension in the EncryptedExtensions, the server indicates the identity of the verifier who issued the attestation results carried in the Certificate message sent by the server. The verifier identity in the results_request extension **MUST** contain a single value selected from the results_request extension in the ClientHello.

9. Background-Check Model Examples

9.1. Cloud Confidential Computing

In this example, a confidential workload is executed on computational resources hosted at a cloud service provider. This is a typical scenario for secure, privacy-preserving multiparty computation, including anti-money laundering, drug development in healthcare, contact tracing in pandemic times, etc.

In such scenarios, the users (e.g., the party providing the data input for the computation, the consumer of the computed attestation results, the party providing a proprietary ML model used in the computation) have two goals:

* Identifying the workload they are interacting with,

* Making sure that the platform on which the workload executes is a Trusted Execution Environment (TEE) with the expected features.

A convenient arrangement is to verify that the two requirements are met at the same time that the secure channel is established.

The protocol flow, alongside all the involved actors, is captured in Figure 12 where the TLS client is the user (the relying party) while the TLS server is co-located with the TEE-hosted confidential workload (the attester).

The flow starts with the client initiating a verification session with a trusted verifier. The verifier returns the evidence types it understands and a nonce that will be used to challenge the attester.

The client starts the TLS handshake with the server by supplying the attestation-related parameters it has obtained from the verifier. If the server supports one of the offered evidence types, it will echo it in the specular extension and proceed by invoking a local API (represented by attest_key(...) in the figure below) to request attestation using the nonce supplied by the verifier. The returned evidence binds the identity key (TIK-S) with the platform identity and security state, packaged into a CAB. The server then signs a transcript hash (represented by hs in the figure below) of the handshake context and the server's Certificate message with the (attested) identity key, and sends the attestation evidence together with the signature over to the client.

The client forwards the attestation evidence to the verifier using the previously established session, obtains the attestation result (AR) and checks whether it is acceptable according to its local policy. If so, it proceeds and verifies the handshake signature

using the corresponding public key (for example, using the PoP key in the KAT evidence [I-D.bft-rats-kat]).

The attestation evidence verification combined with the verification of the CertificateVerify signature provide confirmation that the presented cryptographic identity is bound to the workload and platform identity, and that the workload and platform are trustworthy. Therefore, after the handshake is finalized, the client can trust the workload on the other side of the established secure channel to provide the required confidential computing properties.

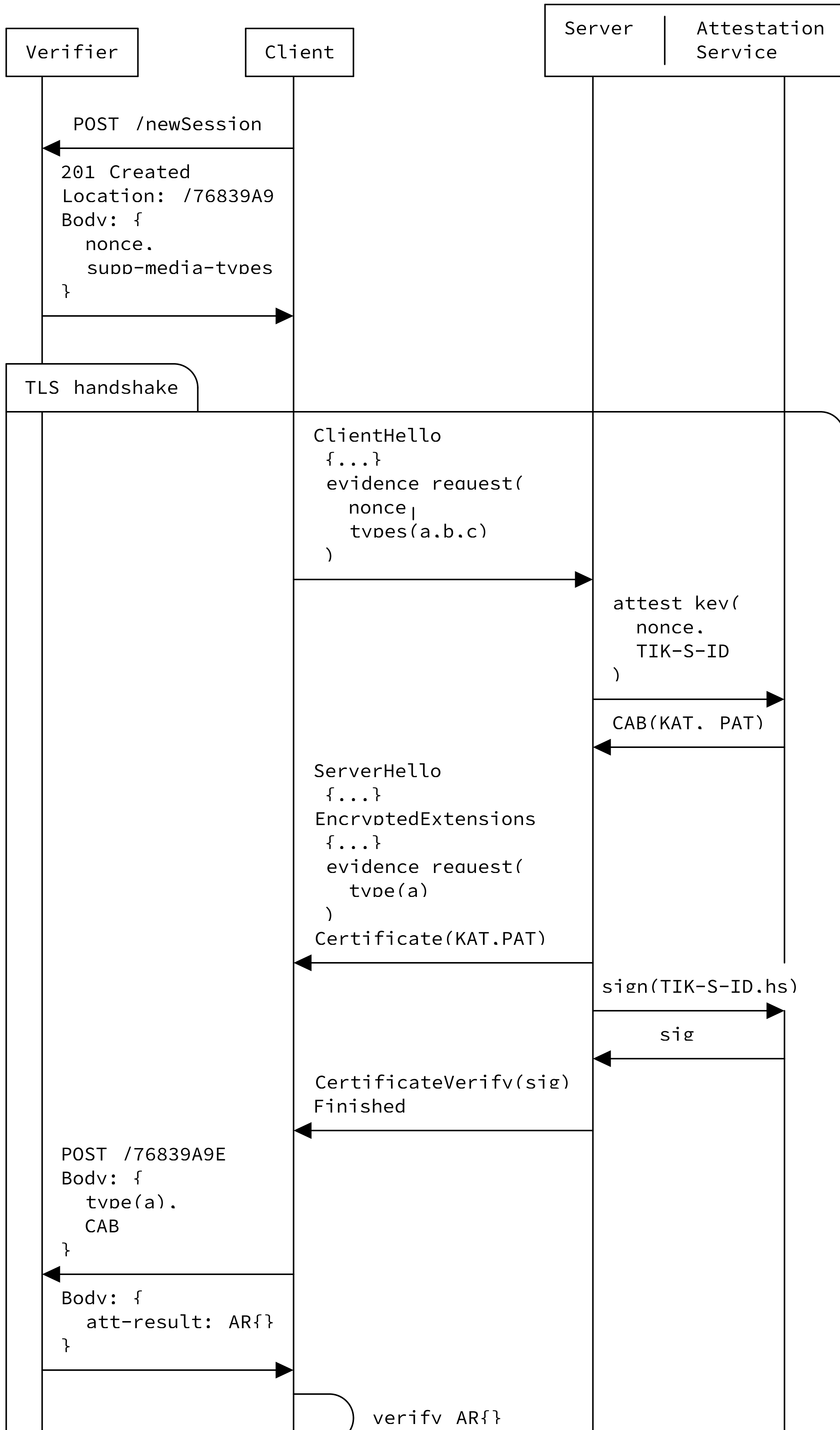

Verifier
Client
Server
Attestation Service
POST /newSession
201 Created
Location: /76839A9
Body: {
nonce,
supp-media-types
}
TLS handshake
ClientHello
{...}
evidence_request(
nonce,
types(a,b,c)
)
attest_key(
nonce,
TIK-S-ID
)
CAB(KAT, PAT)
ServerHello
{...}
EncryptedExtensions
{...}
evidence_request(
type(a)
)
Certificate(KAT,PAT)
sign(TIK-S-ID,hs)
sig
CertificateVerify(sig)
Finished
POST /76839A9E
Body: {
type(a),
CAB
}
Body: {
att-result: AR{}
}
verify AR{}

Figure 12: Example Exchange with Server as Attester.

9.2. IoT Device Onboarding

In this example, an IoT is onboarded to a cloud service provider (or to a network operator). In this scenario there is typically no a priori relationship between the device and the cloud service provider that will remotely manage the device.

In such scenario, the cloud service provider wants to make sure that the device runs the correct version of firmware, has not been rooted, is not emulated or cloned.

The protocol flow is shown in Figure 13 where the client is the attester while the server is the relying party.

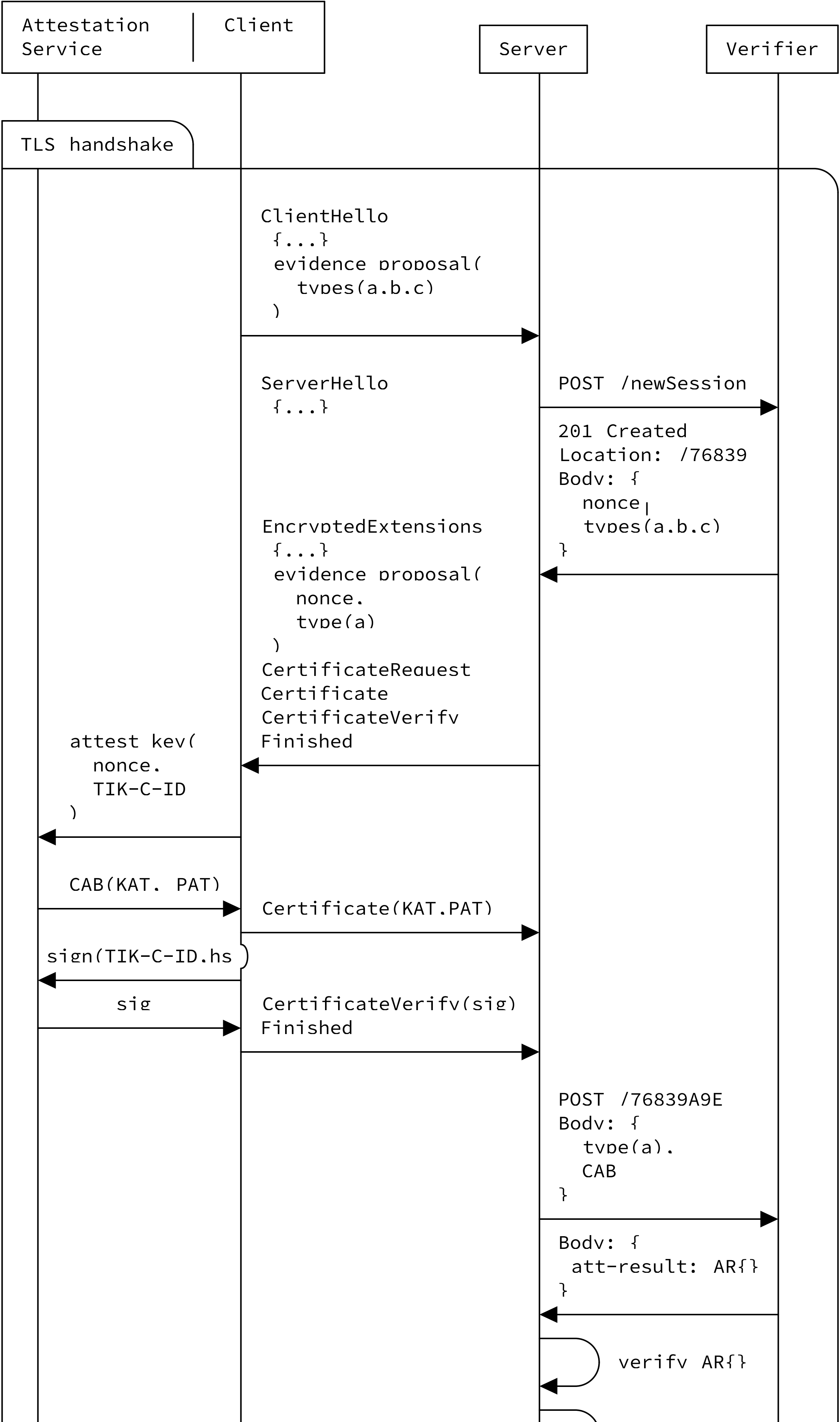
The flow starts with the client initiating a TLS exchange with the TLS server operated by the cloud service provider. The client indicates what evidence types it supports.

The server obtains a nonce from the verifier, in real-time or from a reserved nonce range, and returns it to the client alongside the selected evidence type. Since the evidence will be returned in the Certificate message the server has to request mutual authentication via the CertificateRequest message.

The client, when receiving the EncryptedExtension with the evidence_proposal, will proceed by invoking a local API to request the attestation. The returned evidence binds the identity key (TIK-C) with the workload and platform identity and security state, packaged into a CAB. The client then signs a transcript hash of the handshake context and the client's Certificate message with the (attested) identity key, and sends the evidence together with the signature over to the server.

The server forwards the attestation evidence to the verifier, obtains the attestation result and checks that it is acceptable according to its local policy. The evidence verification combined with the verification of the CertificateVerify signature provide confirmation that the presented cryptographic identity is bound to the platform identity, and that the platform is trustworthy.

If successful, the server proceeds with the application layer protocol exchange. If, for some reason, the attestation result is not satisfactory the TLS server will terminate the exchange.



Attestation Service
Client
Server
Verifier
TLS handshake
ClientHello
{...}
evidence_proposal(
types(a,b,c)
)
ServerHello
{...}
POST /newSession
201 Created
Location: /76839
Body: {
nonce,
types(a,b,c)
}
EncryptedExtensions
{...}
evidence_proposal(
nonce,
type(a)
)
CertificateRequest
Certificate
CertificateVerify
Finished
attest_key(
nonce,
TIK-C-ID
)
CAB(KAT, PAT)
Certificate(KAT,PAT)
sign(TIK-C-ID,hs)
sig
CertificateVerify(sig)
Finished
POST /76839A9E
Body: {
type(a),
CAB
}
Body: {
att-result: AR{}
}
verify AR{}

Figure 13: Example Exchange with Client as Attester.

10. Security Considerations

TBD.

11. IANA Considerations

11.1. TLS Extensions

IANA is asked to allocate four new TLS extensions, evidence_request, evidence_proposal, results_request, results_proposal, from the "TLS ExtensionType Values" subregistry of the "Transport Layer Security (TLS) Extensions" registry [TLS-Ext-Registry]. These extensions are used in the ClientHello and the EncryptedExtensions messages. The values carried in these extensions are taken from TBD.

11.2. TLS Alerts

IANA is requested to allocate a value in the "TLS Alerts" subregistry of the "Transport Layer Security (TLS) Parameters" registry [TLS-Param-Registry] and populate it with the following entries:

- Value: TBD1
- Description: unsupported_evidence
- DTLS-OK: Y
- Reference: [This document]
- Comment:
- Value: TBD2
- Description: unsupported_verifiers
- DTLS-OK: Y
- Reference: [This document]
- Comment:

11.3. TLS Certificate Types

IANA is requested to allocate a new value in the "TLS Certificate Types" subregistry of the "Transport Layer Security (TLS) Extensions" registry [TLS-Ext-Registry], as follows:

- Value: TBD2

*Description: Attestation

*Reference: [This document]

12. References

12.1. Normative References

[I-D.ietf-rats-msg-wrap] Birkholz, H., Smith, N., Fossati, T., and H. Tschofenig, "RATS Conceptual Messages Wrapper (CMW)", Work in Progress, Internet-Draft, draft-ietf-rats-msg-wrap-04, 27 February 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-rats-msg-wrap-04>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[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>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/rfc/rfc8949>.

12.2. Informative References

[DICE-Layering] Trusted Computing Group, "DICE Layering Architecture Version 1.00 Revision 0.19", July 2020, <https://trustedcomputinggroup.org/resource/dice-layering-architecture/>.

[I-D.acme-device-attest]
Weeks, B., "Automated Certificate Management Environment (ACME) Device Attestation Extension", Work in Progress, Internet-Draft, draft-acme-device-attest-02, 22 February 2024, <https://datatracker.ietf.org/doc/html/draft-acme-device-attest-02>.

[I-D.bft-rats-kat] Brossard, M., Fossati, T., and H. Tschofenig, "An EAT-based Key Attestation Token", Work in Progress, Internet-Draft, draft-bft-rats-kat-03, 4 March 2024,

<https://datatracker.ietf.org/doc/html/draft-bft-rats-kat-03>.

[I-D.ietf-rats-ar4si] Voit, E., Birkholz, H., Hardjono, T., Fossati, T., and V. Scarlata, "Attestation Results for Secure Interactions", Work in Progress, Internet-Draft, draft-ietf-rats-ar4si-05, 30 August 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-rats-ar4si-05>.

[I-D.ietf-rats-eat] Lundblade, L., Mandyam, G., O'Donoghue, J., and C. Wallace, "The Entity Attestation Token (EAT)", Work in Progress, Internet-Draft, draft-ietf-rats-eat-25, 15 January 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-rats-eat-25>.

[I-D.ietf-teep-architecture] Pei, M., Tschofenig, H., Thaler, D., and D. M. Wheeler, "Trusted Execution Environment Provisioning (TEEP) Architecture", Work in Progress, Internet-Draft, draft-ietf-teep-architecture-19, 24 October 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-teep-architecture-19>.

[RA-TLS] Knauth, T., Steiner, M., Chakrabarti, S., Lei, L., Xing, C., and M. Vij, "Integrating Remote Attestation with Transport Layer Security", January 2018, <https://arxiv.org/abs/1801.05863>.

[RFC9334] Birkholz, H., Thaler, D., Richardson, M., Smith, N., and W. Pan, "Remote ATtestation procedureS (RATS) Architecture", RFC 9334, DOI 10.17487/RFC9334, January 2023, <https://www.rfc-editor.org/rfc/rfc9334>.

[TLS-Ext-Registry] IANA, "Transport Layer Security (TLS) Extensions", <http://www.iana.org/assignments/tls-extensiontype-values>.

[TLS-Param-Registry] IANA, "Transport Layer Security (TLS) Parameters", <http://www.iana.org/assignments/tls-parameters>.

[TPM1.2] Trusted Computing Group, "TPM Main Specification Level 2 Version 1.2, Revision 116", March 2011, <https://trustedcomputinggroup.org/resource/tpm-main-specification/>.

[TPM2.0] Trusted Computing Group, "Trusted Platform Module Library Specification, Family "2.0", Level 00, Revision 01.59", November 2019, <https://trustedcomputinggroup.org/resource/tpm-library-specification/>.

Appendix A. Design Rationale: X.509 and Attestation Usage Variants

The inclusion of attestation results and evidence as part of the TLS handshake offers the relying party information about the state of the system and its cryptographic keys, but lacks the means to specify a stable endpoint identifier. While it is possible to solve this problem by including an identifier as part of the attestation result, some use cases require the use of a public key infrastructure (PKI). It is therefore important to consider the possible approaches for conveying X.509 certificates and attestation within a single handshake.

In general, the following combinations of X.509 and attestation usage are possible:

1. X.509 certificates only: In this case no attestation is exchanged in the TLS handshake. Authentication relies on PKI alone, i.e. TLS with X.509 certificates.

2. X.509 certificates containing attestation extension: The X.509 certificates in the Certificate message carry attestation as part of the X.509 certificate extensions. Several proposals exist that enable this functionality:

   * Custom X.509 extension:

     - Attester-issued certificates (e.g., RA-TLS [RA-TLS]): The attester acts as a certification authority (CA) and includes the attestation evidence within an X.509 extension.

     - DICE defines extensions that include attestation information in the "Embedded CA" certificates (See Section 8.1.1.1 of [DICE-Layering]).

     - Third party CA-issued certificates (e.g., ACME Device Attestation [I-D.acme-device-attest]): Remote attestation is performed between the third party CA and the attester prior to certificate issuance, after which the CA adds an extension indicating that the certificate key has fulfilled some verification policy.

   * Explicit signalling via existing methods, e.g. using a policy OID in the end-entity certificate.

   * Implicit signalling, e.g. via the issuer name.

3. X.509 certificates alongside a PAT: This use case assumes that a keypair with a corresponding certificate already exists and that the owner wishes to continue using it. As a consequence,

there is no cryptographic linkage between the certificate and the PAT. This approach is described in Section 6.2.

4. X.509 certificates alongside the PAT and KAT: The addition of key attestation implies that the TLS identity key must have been generated and stored securely by the attested platform. Unlike in variant (3), the certificate, the KAT, and the PAT must be cryptographically linked. This variant is currently not addressed in this document.

5. Combined PAT/KAT: With this variant the attestation token carries information pertaining to both platform and key. No X.509 certificate is transmitted during the handshake. This approach is currently not addressed in this document.

6. PAT alongside KAT: This variant is similar to (5) with the exception that the key and the platform attestations are stored in separate tokens, cryptographically linked together. This approach is covered by this document in Section 6.1. A possible instantiation of the KAT is described in [I-D.bft-rats-kat].

Appendix B. Cross-protocol Binding Mechanism

Note: This section describes a protocol-agnostic mechanism which is used in the context of TLS within the body of the draft. The mechanism might, in the future, be spun out into its own document.

One of the issues that must be addressed when using remote attestation as an authentication mechanism is the binding to the outer protocol (i.e., the protocol requiring authentication). For every instance of the combined protocol, the remote attestation credentials must be verifiably linked to the outer protocol. The main reason for this requirement is security: a lack of binding can result in the attestation credentials being relayed.

If the attestation credentials can be enhanced freely and in a verifiable way, the binding can be performed by inserting the relevant data as new claims. If the ways of enhancing the attestation credentials are more restricted, ad-hoc solutions can be devised which address the issue. For example, many roots of trust only allow a small amount (32-64 bytes) of user-provided data which will be included in the attestation token. If more data must be included, it must therefore be compressed. In this case, the problem is compounded by the need to also include a challenge value coming from the relying party. The verification steps also become more complex, as the binding data must be returned from the verifier and checked by the relying party.

However, regardless of how the binding and verification are performed, similar but distinct approaches need to be taken for

every protocol into which remote attestation is embedded, as the type or semantics of the binding data could differ. A more standardised way of tackling this issue would therefore be beneficial. This appendix presents a solution to this problem, in the context of attestation evidence.

B.1. Binding Mechanism

The core of the binding mechanism consists of a new token format - the Attestation Channel Binder - that represents a set of binders as a CBOR map. Binders are individual pieces of data with an unambiguous definition. Each binder is a name/value pair, where the name must be an integer and the value must be a byte string.

Each protocol using the Attestation Channel Binder to bind attestation credentials must define its Attestation Channel Binder using CDDL. The only mandated binder is the challenger nonce which must use the value 1 as a name. Every other name/value pair must come with a text description of its semantics. The byte strings forming the values of binders can be size-restricted where this value is known.

Attestation Channel Binders are encoded in CBOR, following the CBOR core deterministic encoding requirements (Section 4.2.1 of [RFC8949]).

An example Attestation Channel Binder is shown below.

```
attestation_channel_binder = {
  &(nonce: 1) => bstr .size (8..64)
  &(ik_pub_fingerprint: 2) => bstr .size 32
  &(session_key_binder: 3) => bstr .size 32
}
```

Figure 14: Format of a possible TLS Attestation Channel Binder.

B.2. Usage

When a Attestation Channel Binder is used to compress data to fit the space afforded by an attestation scheme, the encoded binder must be hashed. Since the relying party has access to all the data expected in the binder, the binder itself need not be conveyed. How the hashing algorithm is chosen, used, and conveyed must be defined per outer protocol. If the digest size does not match the user data size mandated by the attestation scheme, the digest is truncated or expanded appropriately.

The verifier must first hash the encoded token received from the relying party and then compare the hashes. The challenge value

included in the binder can then be extracted and verified. If verification is successful, binder correctness can also be assumed by the relying party, as verification was done with the values it expected.

Appendix C. History

RFC EDITOR: PLEASE REMOVE THIS SECTION

C.1. draft-fossati-tls-attestation-02

* Focus on the background check model

* Added examples

* Updated introduction

* Moved attestation format-specific content to related drafts.

C.2. draft-fossati-tls-attestation-01

* Added details about TPM attestation

C.3. draft-fossati-tls-attestation-00

* Initial version

Appendix D. Working Group Information

The discussion list for the IETF TLS working group is located at the e-mail address tls@ietf.org. Information on the group and information on how to subscribe to the list is at https://www1.ietf.org/mailman/listinfo/tls

Archives of the list can be found at: https://www.ietf.org/mail-archive/web/tls/current/index.html

Authors' Addresses

Hannes Tschofenig

Email: hannes.tschofenig@gmx.net

Yaron Sheffer
Intuit

Email: yaronf.ietf@gmail.com

Paul Howard
Arm Limited

Email: Paul.Howard@arm.com

Ionut Mihalcea
Arm Limited

Email: Ionut.Mihalcea@arm.com

Yogesh Deshpande
Arm Limited

Email: Yogesh.Deshpande@arm.com

Arto Niemi
Huawei

Email: arto.niemi@huawei.com