Workgroup: TLS
Internet-Draft:
draft-fossati-tls-attestation-01
Published: 26 August 2022
Intended Status: Standards Track
Expires: 27 February 2023
A H. Tschofenig T. Fossati P. Howard
uArm Limited Arm Limited Arm Limited
t
h
o
r
s
:
I. Mihalcea Y. Deshpande
Arm Limited Arm Limited

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

Abstract

Various attestation technologies have been developed and formats have been standardized. Examples include the Entity Attestation Token (EAT) and Trusted Platform Modules (TPMs). Once attestation information has been produced on a device it needs to be communicated to a relying party. This information exchange may happen at different layers in the protocol stack.

This specification provides a generic way of passing attestation information in the TLS handshake.

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 27 February 2023.

Table of Contents

1. Introduction

Attestation is the process by which an entity produces evidence about itself that another party can use to evaluate the trustworthiness of that entity. One format of encoding evidence is standardized with the Entity Attestation Token (EAT) [I-D.ietf-rats-eat] but there are other formats, such as attestation produced by Trusted Platform Modules (TPMs) [TPM1.2] [TPM2.0].

This specification defines how to convey attestation information in the TLS handshake with different encodings being supported. This specification standardizes two attestation formats - EAT and TPM-based attestation.

Note: This initial version of the specification focuses on EAT-based attestation. Future versions will also define TPM-based attestation.

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 RFC 2119 [RFC2119].

The following terms are used in this document:

- Root of Trust (RoT): A set of software and/or hardware components that need to be trusted to act as a security foundation required for accomplishing the security goals. In our case, the RoT is expected to offer the functionality for attesting to the state of the platform.

- Attestation Key (AK): Cryptographic key belonging to the RoT that is only used to sign attestation tokens.

- Platform Attestation Key (PAK): An AK used specifically for signing attestation tokens relating to the state of the platform.

- Key Attestation Key (KAK): An AK used specifically for signing KATs. In some systems only a single AK is used. In that case the AK is used as a PAK and a KAK.

- TLS Identity Key (TIK): The KIK consists of a private and a public key. The private key is used in the CertificateVerify message during the TLS handshake. The public key is included in the Key Attestation Token.

- Attestation Token (AT): A collection of claims that a RoT assembles (and signs) with the purpose of informing - in a verifiable way - Relying Parties about the identity and state of the platform. Essentially a type of "Evidence" as per the RATS architecture terminology.

- Platform Attestation Token (PAT): An AT containing claims relating to the state of the software running on the platform. The process of generating a PAT typically involves gathering data during measured boot.

- Key Attestation Token (KAT): An AT containing a claim with a proof-of-possession (PoP) key. The KAT may also contain other claims, such as those indicating its validity. The KAT is signed by the KAK. The attestation service part of the RoT conceptually acts as a local certification authority since the KAT behaves like a certificate.

- Combined Attestation Bundle (CAB): A structure used to bundle a KAT and a PAT together for transport in the TLS handshake. If the KAT already includes a PAT, in form of a nested token, then it already corresponds to a CAB.

3. Overview

The Remote Attestation Procedures (RATS) architecture [I-D.ietf-rats-architecture] defines two types of interaction models for attestation, namely the passport model and the background-check model. The subsections below explain the difference in their interactions.

To simplify the description in this section we focus on the use case where the client is the attester and the server is the relying party. Hence, only the client_attestation_type extension is discussed. The verifier is not shown in the diagrams. The described mechanism allows the roles to be reversed.

As typical with new features in TLS, the client indicates support for the new extension in the ClientHello. The client_attestation_type extension lists the supported attestation formats. The server, if it supports the extension and one of the attestation formats, it confirms the use of the feature.

Note: The newly introduced extension also allows nonces to be exchanged. Those nonces are used for guaranteeing freshness of the generated attestation tokens.

When the attestation extension is successfully negotiated, the content of the Certificate message is replaced with attestation information described in this document.

A peer has to demonstrate possession of the private key via the CertificateVerify message. While attestation information is signed by the attester, it typically does not contain a public key (for example via a proof-of-possession key claim [RFC8747]).

The attestation service on a device, which creates the attestation information, is unaware of the TLS exchange and the attestation service does not directly sign externally provided data, as it would be required to compute the CertificateVerify message.

Hence, the following steps happen:

The client generates the TIK, which are referred here as skT and pkT, for example using the following API call:

```
key_id = GenerateKeyPair(alg_id)
```

The private key would be created and stored by the crypto hardware supported by the device (rather than the TLS client in software).

Next, the attestation service needs to be triggered to create a Platform Attestation Token (PAT) and the Key Attestation Token (KAT). The Key Attestation Token (KAT) includes a claim containing the public key of the TIK (pkT). The KAT is then signed with the Key Attestation Key (KAK).

To ensure freshness of the PAT and the KAT a nonce is provided by the relying party / verifier. Here is the symbolic API call to request a KAT and a PAT, which are concatinated together as the CAB.

```
cab = createCAB(key_id, nonce)
```

Once the Certificate message containing the CAB has been sent, the CertificateVerify has to be created and it requires access to the private key. The signature operation uses the private key of the TIK (skT).

The recipient of the Certificate and the CertificateVerify messages first extracts the PAT and the KAT from the Certificate message. The PAT and the KAT need to be conveyed to the verification service, whereby the following checks are made:

- The signature protecting the PAT passes verification when using available trust anchor(s).
- The PAT has not been replayed, which can be checked by comparing the nonce included in one of the claims and matching it against the nonce provided to the attester.
- The claims in the PAT are matched against stored reference values.
- The signature protecting the KAT passes verification.
- The claims in the KAT are validated, if needed.

Once all these steps are completed, the verifier produces the attestation result and includes (if needed) the TIK public key.

In the subsections we will look at how the two message pattern fit align with the TLS exchange.

3.1. Attestation within the Passport Model

The passport model is described in Section 5.1 of [I-D.ietf-rats-architecture]. A key feature of this model is that the attester interacts with the verification service before initiating the TLS exchange. It sends evidence to the verification service, which then returns the attestation result (including the TIK public key).

The example exchange in Figure 1 shows how a client provides attestation to the server by utilizing EAT tokens [I-D.ietf-rats-eat]. With the ClientHello the TLS client needs to indicate that it supports the EAT-based attestation format. The TLS server acknowledges support for this attestation type in the EncryptedExtensions message.

In the Certificate message the TLS client transmits the attestation result to the TLS server, in form a CAB (i.e. a concatinated PAT and KAT).

The TLS client then creates the CertificateVerify message by asking the crypto service to sign the TLS handshake message transcript with the TIK private key. The TLS server then verifies this message by utilizing the TIK public key.

Figure 1: Example Exchange with the Passport Model.

3.2. Attestation within the Background Check Model

The background check model is described in Section 5.2 of [I-D.ietf-rats-architecture].

The message exchange of the background check model differs from the passport model because the TLS server needs to provide a nonce in the ServerHello to the TLS client so that the attestation service can feed the nonce into the generation of the PAT. The TLS server, when receiving the CAB, will have to contact the verification service.

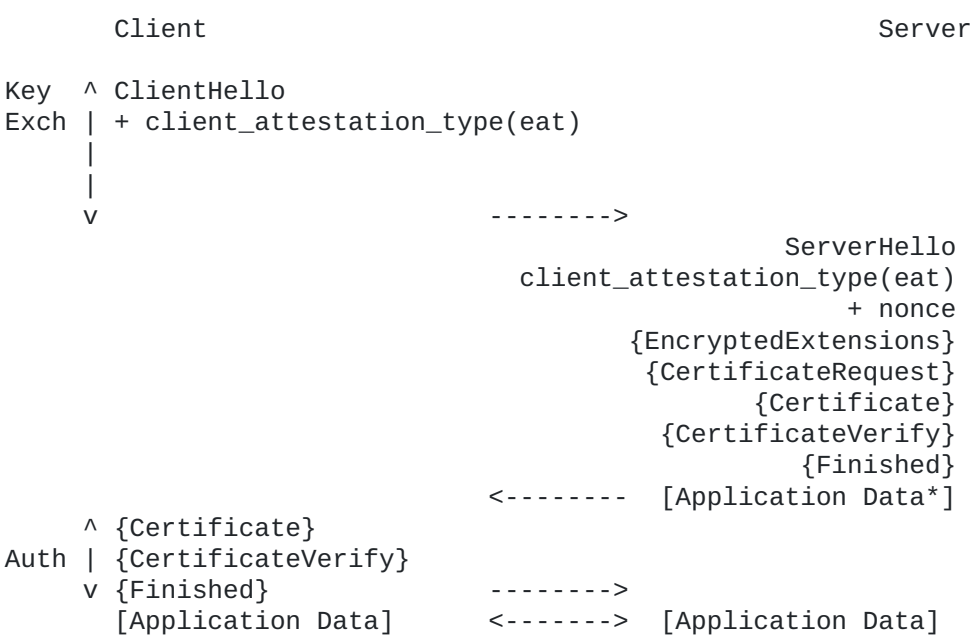

Figure 2: Example Exchange with the Background Check Model.

4. TLS Attestation Type Extension

This document defines a new extension to carry the attestation types. The extension is conceptually similiar to the 'server_certificate_type' and the 'server_certificate_type' defined by [RFC7250].

```
struct {
        select(ClientOrServerExtension) {
            case client:
              CertificateType client_attestation_types<1..2^8-1>;
              opaque nonce<0..2^16-1>;

            case server:
              CertificateType client_attestation_type;
              opaque nonce<0..2^16-1>;
        }
} ClientAttestationTypeExtension;

struct {
        select(ClientOrServerExtension) {
            case client:
              CertificateType server_attestation_types<1..2^8-1>;
              opaque nonce<0..2^16-1>;

            case server:
              CertificateType server_attestation_type;
              opaque nonce<0..2^16-1>;
        }
} ServerAttestationTypeExtension;
```

Figure 3: AttestationTypeExtension Structure.

The Certificate payload is used as a container, as shown in Figure 4. The shown Certificate structure is an adaptation of [RFC8446].

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

          /* attestation type defined in this document */
        case EAT:
          opaque cab<1..2^24-1>;

          /* attestation type defined in this document */
        case TPM:
          opaque tpmStmtFormat<1..2^24-1>;

        case X509:
          opaque cert_data<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 4: Certificate Message.

To simplify parsing of an EAT-based attestation payload, the PAT and the KAT are typed.

5. TLS Client and Server Handshake Behavior

This specification extends the ClientHello and the EncryptedExtensions messages, according to [RFC8446].

The high-level message exchange in Figure 5 shows the client_attestation_type and server_attestation_type extensions added to the ClientHello and the EncryptedExtensions messages.

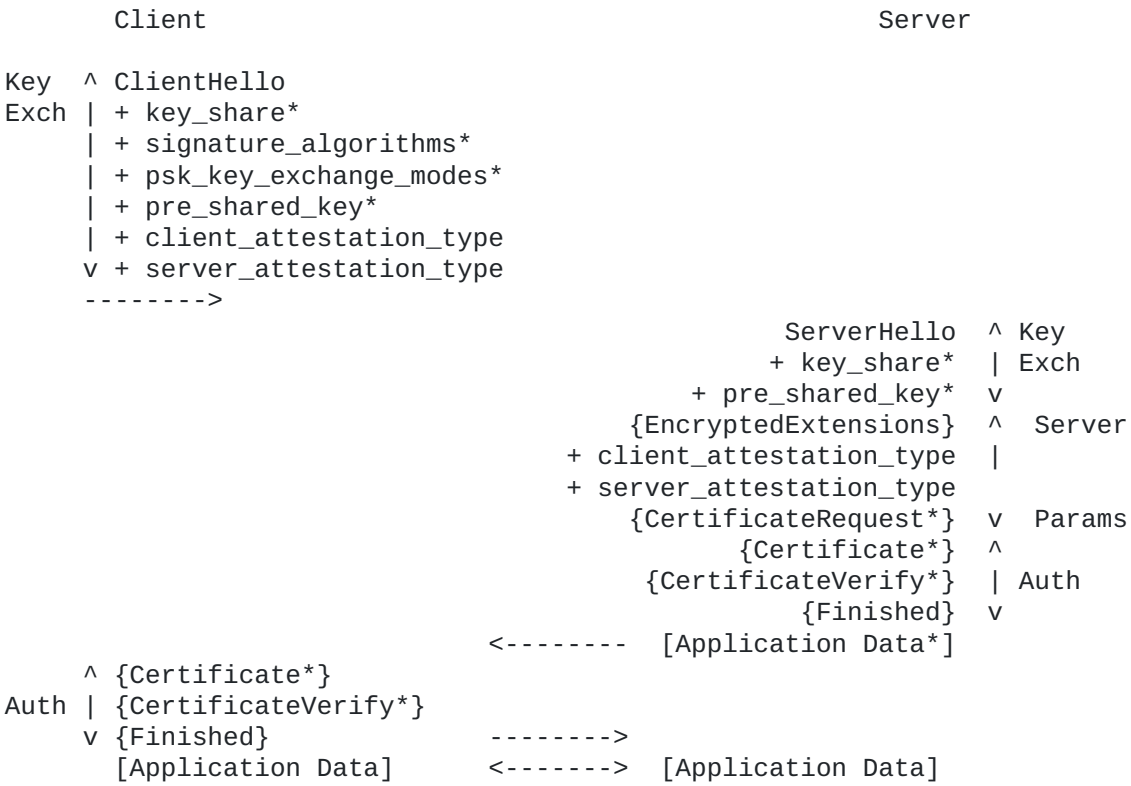

Figure 5: Attestation Message Overview.

5.1. Client Hello

In order to indicate the support of attestation types, clients include the client_attestation_type and/or the server_attestation_type extensions in the ClientHello.

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

The server_attestation_type extension in the ClientHello indicates the types of attestation types the client is able to process when provided by the server in a subsequent Certificate payload.

The client_attestation_type and server_attestation_type extensions sent in the ClientHello each carry a list of supported attestation types, sorted by client preference. When the client supports only one attestation type, it is a list containing a single element.

The TLS client MUST omit attestation types from the client_attestation_type extension in the ClientHello if it is not equipped with the corresponding attestation functionality, or if it is not configured to use it with the given TLS server. If the client has no attestation types to send in the ClientHello it MUST omit the client_attestation_type extension in the ClientHello.

The TLS client MUST omit attestation types from the server_attestation_type extension in the ClientHello if it is not equipped with the attestation verification functionality. If the client has no attestation types to send in the ClientHello it MUST

omit the entire server_attestation_type extension from the ClientHello.

5.2. Server Hello

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

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

- The server supports the extension defined in this document, but it does not have any attestation type in common with the client. Then, the server terminates the session with a fatal alert of type "unsupported_certificate".

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

The client_attestation_type extension in the ClientHello indicates the attestation types the client is able to provide to the server, when requested using a certificate_request message. If the TLS server wants to request a certificate from the client (via the certificate_request message), it MUST include the client_attestation_type extension in the EncryptedExtensions. This client_attestation_type extension in the EncryptedExtensions then indicates the content the client is requested to provide in a subsequent Certificate payload. The value conveyed in the client_attestation_type extension MUST be selected from one of the values provided in the client_attestation_type extension sent in the client hello. The server MUST also include a certificate_request payload in the EncryptedExtensions message.

If the server does not send a certificate_request payload (for example, because client authentication happens at the application layer or no client authentication is required) or none of the attestation types supported by the client (as indicated in the client_attestation_type extension in the ClientHello) match the server-supported attestation types, then the client_attestation_type payload in the ServerHello MUST be omitted.

The server_attestation_type extension in the ClientHello indicates the types of attestation types the client is able to process when provided by the server in a subsequent Certificate message. With the server_attestation_type extension in the EncryptedExtensions, the TLS server indicates the attestation type carried in the Certificate payload. Note that only a single value is permitted in the server_attestation_type extension when carried in the EncryptedExtensions message.

6. TPM Attestation

The Trusted Platform Module (TPM) [TPM2.0] is one type of hardware RoT. TPMs offer the ability to produce both key and platform attestation tokens.

6.1. Platform Attestation

Platform Configuration Registers (PCRs) represent the core mechanism in TPMs for measuring and conveying information about the platform state via remote attestation. While specifications exist for assigning individual PCRs to specific software components, the choice of which combination of PCRs to include for any attestation procedure (and which hashing algorithm to use) is left to the parties involved. The agreement over and the configuration of the PCR selection falls outside the scope of this specification and is thus expected to occur out-of-band.

The attestation evidence is produced through the TPM2_Quote operation. The evidence along with all other relevant metadata is transmitted in a format derived from the [WebAuthn] Attestation Statements. This format and the workflows around it are defined below.

6.1.1. TPM Platform Attestation Statement Format

The TPM Platform Attestation Statement is a modified version of the TPM Attestation Statement Format, which covers key attestation tokens.

```
tpmPlatStmtFormat = {
                      ver: "2.0",
                      (
                          alg: COSEAlgorithmIdentifier,
                          x5c: [ pakCert: bytes, * (caCert: bytes) ]
                      )
                      sig: bytes,
                      attestInfo: bytes,
                    }
```

Figure 6: TPM Platform Attestation Statement Format

* *ver*: The version of the TPM specification to which the signature conforms.

* *alg*: A COSEAlgorithmIdentifier containing the identifier of the algorithm used to generate the attestation signature.

* *x5c*: A certificate for the PAK, followed by its certificate chain. The contents of the array SHOULD follow the same requirements as the *x5chain* header parameter defined in Section 2 of [I-D.ietf-cose-x509], with the sole difference that a CBOR array is also used when only *pakCert* is present.

  - *pakCert*: The PAK certificate used for the attestation.

*_sig_: The attestation signature, in the form of a TPMT_SIGNATURE structure as specified in Part 2, Section 11.3.4 of [TPM2.0].

*_attestInfo_: The TPMS_ATTEST structure over which the above signature was computed, as specified in Part 2, Section 10.12.8 of [TPM2.0].

6.1.2. Signing Procedure

Generate a signature using the operation specified in Part 3, Section 18.4 of [TPM2.0], using the PAK as the signing key, the out-of-band agreed-upon PCR selection. Freshness of the attestation is given by the nonce provided by the relying party. The nonce is included as qualified data to the TPM2_Quote operation, concatenated with an identifier of the platform being attested, as shown below:

extraData = _platformUuid_ || _relyingPartyNonce_

The platform identifier is a 16-bytes long UUID, with the remaining data representing the nonce. The UUID is intended to help the verifier link the platform with its expected reference values.

Set the _attestInfo_ field to the quoted PCR selection produced by the operation, and _sig_ to the signature generated above.

6.1.3. Verification Procedure

The inputs to the verification procedure are as follows:

*an attestation statement in the format described above

*the nonce sent to the attester

*a database of reference values for various platforms

The steps for verifying the attestation:

*Verify that the attestation token is a valid CBOR structure conforming to the CTAP2 canonical CBOR encoding form defined in Section 6 of [CTAP2], and perform CBOR decoding on it to extract the contained fields.

*Verify that _alg_ describes a valid, accepted signing algorithm.

*Verify that _x5c_ is present and follows the requirements laid out for _x5chain_ in [I-D.ietf-cose-x509].

*Verify the _sig_ is a valid signature over _attestInfo_ using the attestation public key in _pakCert_ with the algorithm specified in _alg_.

*Verify that _pakCert_ meets the requirements in Section 8.3.1 of [WebAuthn].

*Verify that _attestInfo_ is valid:

  -Verify that _magic_ is set to TPM_GENERATED_VALUE.

-Verify that *type* is set to TPM_ST_ATTEST_QUOTE.

-Verify that *attested* contains a TPMS_QUOTE_INFO structure as specified in Part 2, Section 10.12.4 of [TPM2.0].

-Extract *extraData* and parse it assuming the format defined above to obtain platform UUID and the nonce. Verify that the nonce is correct.

-Verify that the platform UUID obtained earlier is valid and represents a platform found in the database.

-Retrieve the reference values defined for this platform. Compute the digest of the concatenation of all relevant PCRs using the hash algorithm defined in *alg*. The PCRs are concatenated as described in "Selecting Multiple PCR" (Part 1, Section 17.5 of [TPM2.0]). Verify that this digest is equal to *pcrDigest* in *attested* and that the hash algorithm defined in *pcrSelect* is aligned with the one in *alg*.

-Note that the remaining fields in the "Standard Attestation Structure" (Part 1, Section 31.2 of [TPM2.0]), i.e., *qualifiedSigner*, *clockInfo* and *firmwareVersion* are ignored. These fields MAY be used as an input to risk engines.

*If successful, return implementation-specific values representing attestation type AttCA and attestation trust path *x5c*.

6.2. Key Attestation

Attesting to the provenance and properties of a key is possible through a TPM if the key resides on the TPM. The TPM 2.0 key attestation mechanism used in this specification is TPM2_Certify. The workflow for generating the evidence and assessing them, as well as the format used to transport them, follows closely the TPM Attestation Statement defined in Section 8.3 of [WebAuthn], with one modification:

*For both signing and verification, *attToBeSigned* is unnecessary and therefore its hash is replaced with the nonce coming from the relying party as the qualifying data when signing, and as the expected *extraData* value during verification.

The WebAuthn specification [WebAuthn] uses the term AIK to refer to the signing key. In this specification we use the term KAK instead. The credential (i.e., attested) key is in our case the TIK.

7. Security Considerations

TBD.

8. IANA Considerations

TBD: Create new registry for attestation types.

9. References

9.1. Normative References

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

[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>.

9.2. Informative References

[CTAP2] World Wide Web Consortium, "Client to Authenticator Protocol", January 2019, <https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html>.

[I-D.ietf-cose-x509] Schaad, J., "CBOR Object Signing and Encryption (COSE): Header parameters for carrying and referencing X.509 certificates", Work in Progress, Internet-Draft, draft-ietf-cose-x509-08, 14 December 2020, <https://www.ietf.org/archive/id/draft-ietf-cose-x509-08.txt>.

[I-D.ietf-rats-architecture] Birkholz, H., Thaler, D., Richardson, M., Smith, N., and W. Pan, "Remote Attestation Procedures Architecture", Work in Progress, Internet-Draft, draft-ietf-rats-architecture-21, 16 August 2022, <https://www.ietf.org/archive/id/draft-ietf-rats-architecture-21.txt>.

[I-D.ietf-rats-eat] Lundblade, L., Mandyam, G., and J. O'Donoghue, "The Entity Attestation Token (EAT)", Work in Progress, Internet-Draft, draft-ietf-rats-eat-14, 10 July 2022, <https://www.ietf.org/archive/id/draft-ietf-rats-eat-14.txt>.

[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>.

[RFC8747] Jones, M., Seitz, L., Selander, G., Erdtman, S., and H. Tschofenig, "Proof-of-Possession Key Semantics for CBOR Web Tokens (CWTs)", RFC 8747, DOI 10.17487/RFC8747, March 2020, <https://www.rfc-editor.org/info/rfc8747>.

[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/>.

[WebAuthn] World Wide Web Consortium, "Web Authentication: An API for accessing Public Key Credentials, Level 2", April 2021, <https://www.w3.org/TR/webauthn/>.

Appendix A. History

RFC EDITOR: PLEASE REMOVE THE THIS SECTION

*Initial version

Appendix B. 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
Arm Limited

Email: hannes.tschofenig@arm.com

Thomas Fossati
Arm Limited

Email: Thomas.Fossati@arm.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