Workgroup: TLS
Internet-Draft:
draft-fossati-tls-attestation-03
Published: 13 March 2023
Intended Status: Standards Track
Expires: 14 September 2023
Authors: H. Tschofenig Y. Sheffer P. Howard
Intuit Arm Limited
I. Mihalcea Y. Deshpande
Arm Limited Arm Limited

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

Abstract

Attestation is the process by which an entity produces evidence about itself that another party can use to evaluate the trustworthiness of that entity.

In use cases that require the use of remote attestation, such as confidential computing or device onboarding, an attester has to convey evidence or attestation results to a relying party. This information exchange may happen at different layers in the protocol stack.

This specification provides a generic way of passing evidence and attestation results in the TLS handshake. Functionality-wise this is accomplished with the help of key attestation.

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

Copyright Notice

Table of Contents

1. Introduction

The Remote ATtestation ProcedureS (RATS) architecture defines two basic types of topological patterns to communicate between an attester, a relying party, and a verifier, namely the background-check model and the passport model. These two models are fundamentally different and require a different treatment when incorporated into the TLS handshake. For better readability we suggest to use different extensions for these two models.

The two models can be summarized as follows:

- In the background check model, the attester conveys evidence to the relying party, which then forwards the evidence to the verifier for appraisal; the verifier computes the attestation result and sends it back to the relying party.
- In the passport model, the attester transmits evidence to the verifier directly and receives attestation results, which are then relayed to the relying party.

This specification supports both patterns.

Several formats for encoding evidence are available, such as: - the Entity Attestation Token (EAT) [I-D.ietf-rats-eat], - the Trusted Platform Modules (TPMs) [TPM1.2] [TPM2.0], - the Android Key Attestation, and - Apple Key Attestation.

Likewise, there are different encodings available for attestation results. One such encoding, AR4SI [I-D.ietf-rats-ar4si] is being standardized by the RATS working group.

This version of the specification defines how to support the background check model in the TLS handshake, such that the details about the attestation technology are agnostic to the TLS handshake itself. Later versions of the specification will support the passport model as well.

To give the peer information that the handshake signing key is properly secured, the associated evidence has to be verified by that peer. Hence, attestation evidence about the security state of the signing key is needed, which is typically associated with evidence about the overall platform state. The platform attestation service ensures that the key attestation service has not been tampered with. The platform attestation service issues the Platform Attestation Token (PAT) and the key attestation service issues the Key Attestation Token (KAT). The security of the protocol critically depends on the verifiable binding between these two logically separate units of evidence.

This document does not define how different attestation technologies are encoded. This is accomplished by companion specifications.

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].

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.

As typical with new features in TLS, the client indicates support for the new extension in the ClientHello message. The newly introduced extensions allow evidence and nonces to be exchanged. The nonces are used for guaranteeing freshness of the exchanged evidence.

When the evidence extension is successfully negotiated, the content of the Certificate message contains a payload that is encoded based on the wrapper defined in [I-D.ftbs-rats-msg-wrap].

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 a key attestation token, which associates a private key with the attestation information. An example of a key attestation token format utilizing the EAT format can be found in [I-D.bft-rats-kat].

The recipient extracts evidence from the Certificate message and relays it to the verifier to obtain attestation results. Subsequently, the attested key is used to verify the CertificateVerify message.

4. Use of Evidence with the Background Check Model

The background check model is described in Section 5.2 of [I-D.ietf-rats-architecture] and allows the following modes of operation 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 three cases in sub-sections below.

4.1. TLS Client as Attester

In this use case the TLS client, as the attester, is challenged by the TLS server to provide evidence. The TLS client is the attester and the the TLS server acts as a relying party. 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.

```
       Client                                           Server

Key  ^ ClientHello
Exch | + evidence_proposal
     | + key_share*
     | + signature_algorithms*
     v                         -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                                               v
                                        {EncryptedExtensions}  ^  Server
                                          + evidence_proposal  |  Params
                                                      (nonce)  |
                                         {CertificateRequest}  v
                                                {Certificate}  ^
                                          {CertificateVerify}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate}
Auth | {CertificateVerify}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

Figure 1: TLS Client Providing Evidence to TLS Server.

4.2. TLS Server as Attester

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.

```
       Client                                           Server

Key  ^ ClientHello
Exch | + evidence_request
     |   (nonce)
     | + key_share*
     | + signature_algorithms*
     v                         -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                                               v
                                        {EncryptedExtensions}  ^  Server
                                          + evidence_request   |  Params
                                                               |
                                         {CertificateRequest}  v
                                                {Certificate}  ^
                                          {CertificateVerify}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate}
Auth | {CertificateVerify}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

Figure 2: TLS Client Providing Evidence to TLS Server.

5. Evidence Extensions (Background Check Model)

This document defines two new extensions, the evidence_request and the evidence_proposal, for use with the background check model.

The EvidenceType structure encodes either a media type or as a content format. The media type is a string-based identifier while the content format uses a number. The former is more flexible and does not necessarily require a registration through IANA while the latter is more efficient over-the-wire.

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 { NUMERIC(0), STRING(1) } encodingType;
enum { ATTESTATION(0), CERT_ATTESTATION(1) } credentialType;

struct {
     encodingType type;
     credentialType cred_type;
     select (encodingType) {
         case NUMERIC:
           uint16 content_format;
         case STRING:
            opaque media_type<0..2^16-1>;
     };
} EvidenceType;

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

            case server:
              EvidenceType selected_evidence_type;
        }
} evidenceRequestTypeExtension;

struct {
        select(ClientOrServerExtension) {
            case client:
              EvidenceType supported_evidence_types<1..2^8-1>;

            case server:
              EvidenceType selected_evidence_type;
              opaque nonce<0..2^8-1>;
        }
} evidenceProposalTypeExtension;
```

Figure 3: TLS Structure for Evidence.

5.1. 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 4, 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>;

          /* payload used to convey evidence */
        case attestation:
          opaque evidence<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 when using only attestation.

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

5.2. 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 5.

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

        /* X.509 certificate conveyed as usual */
        case X509:
          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 5: Certificate Message when using PKIX and attestation.

The encoding of the evidence structure is defined in [I-D.ftbs-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 6.

```
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 6: Format of TLS channel binder.

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

- The identity key public 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 partial transcript of the TLS handshake, up to (but not including) the Certificate message.

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.

6. TLS Client and Server Handshake Behavior

The high-level message exchange in Figure 7 shows the evidence_proposal and evidence_request extensions added to the ClientHello and the EncryptedExtensions messages.

```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     | + pre_shared_key*
     | + evidence_proposal
     v + evidence_request
     -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                          + evidence_proposal  |
                                           + evidence_request  |
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

Figure 7: Attestation Message Overview.

6.1. 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.

6.2. 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 client_attestation_type 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.

7. Background-Check Model Examples

7.1. 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 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 8 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 kinds of evidence 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 the local API to request the attestation. The returned evidence binds the identity key with the platform identity and security state. The server then signs the handshake transcript 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 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.

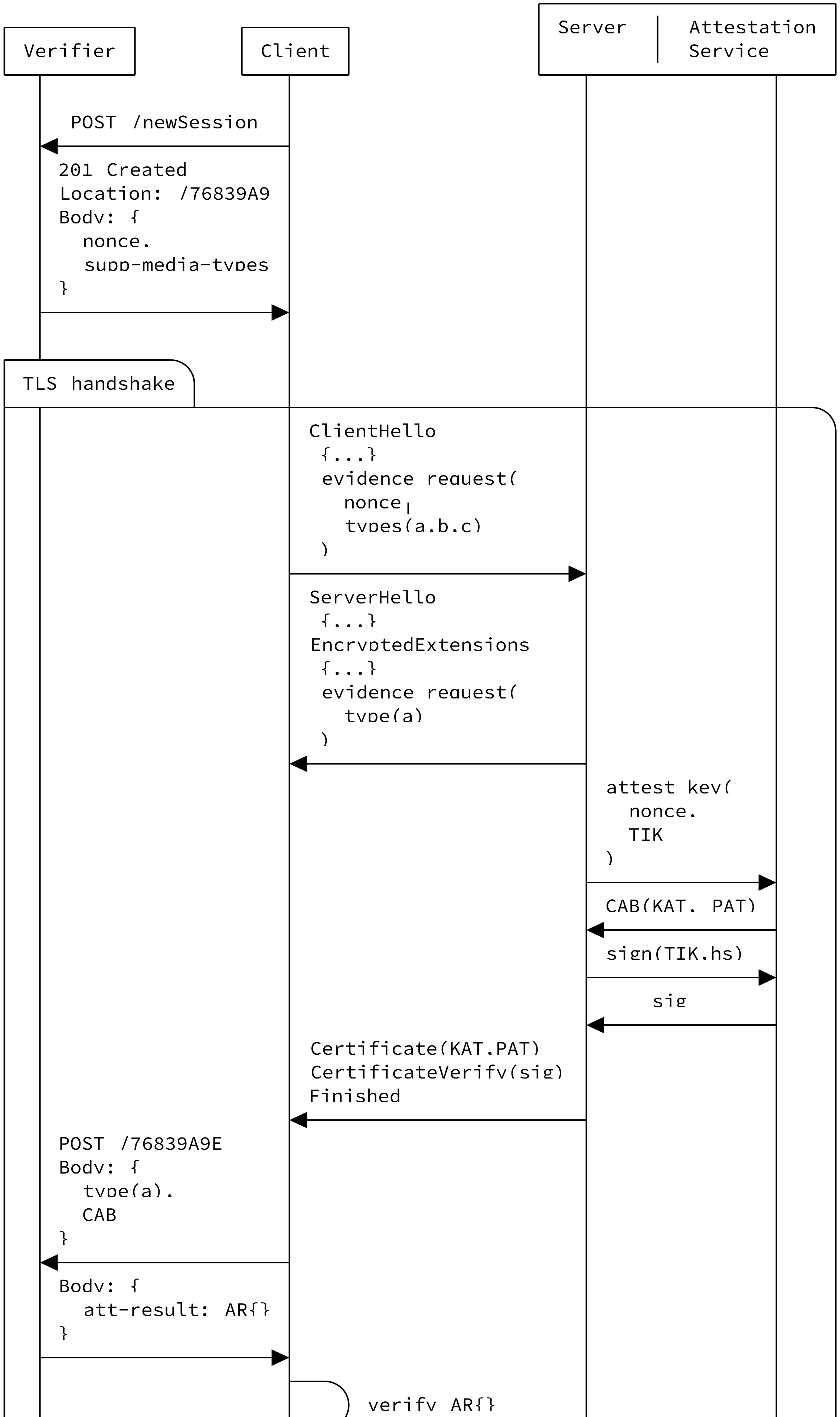
Verifier
Client
Server
Attestation Service
POST /newSession
201 Created
Location: /76839A9
Body: {
nonce,
supp-media-types
}
TLS handshake
ClientHello
{...}
evidence request(
nonce,
types(a,b,c)
)
ServerHello
{...}
EncryptedExtensions
{...}
evidence request(
type(a)
)
attest key(
nonce,
TIK
)
CAB(KAT,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

Figure 8: Example Exchange with Server as Attester.

7.2. 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 9 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 with the workload and platform identity and security state. The client then signs the handshake transcript 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.

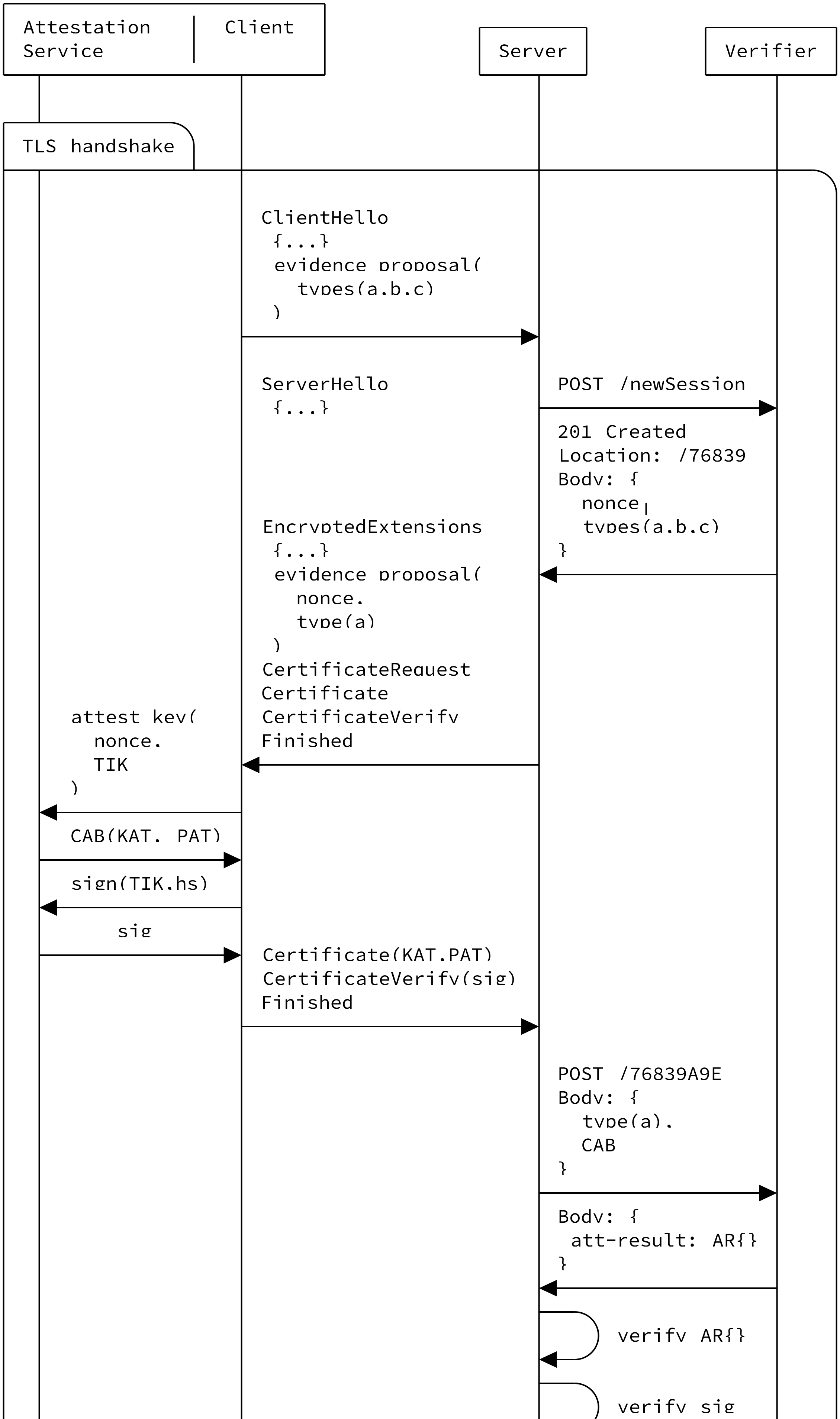
Attestation Service
Client
Server
Verifier
TLS handshake
ClientHello
{...}
evidence_proposal(
types(a,b,c)
)
ServerHello
{...}
POST /newSession
201 Created
Location: /76839
Body: {
nonce,
types(a,b,c)
}
EncryptedExtensions
{...}
evidence_proposal(
nonce,
type(a)
)
CertificateRequest
Certificate
CertificateVerify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

Figure 9: Example Exchange with Client as Attester.

8. Security Considerations

TBD.

9. IANA Considerations

9.1. TLS Extensions

IANA is asked to allocate two new TLS extensions, evidence_request and evidence_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.

9.2. 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 entry:

*Value: TBD1

*Description: unsupported_evidence

*DTLS-OK: Y

*Reference: [This document]

*Comment:

9.3. 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]

10. References

10.1. Normative References

[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-00, 21 October 2022, <https://datatracker.ietf.org/doc/html/draft-bft-rats-kat-00>.

[I-D.ftbs-rats-msg-wrap] Birkholz, H., Smith, N., Fossati, T., and H. Tschofenig, "RATS Conceptual Messages Wrapper", Work in Progress, Internet-Draft, draft-ftbs-rats-msg-wrap-02, 7 March 2023, <https://datatracker.ietf.org/doc/html/draft-ftbs-rats-msg-wrap-02>.

[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>.

[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>.

10.2. 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-00, 12 December 2022, <https://datatracker.ietf.org/doc/html/draft-acme-device-attest-00>.

[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-04, 2 March 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-rats-ar4si-04>.

[I-D.ietf-rats-architecture] Birkholz, H., Thaler, D., Richardson, M., Smith, N., and W. Pan, "Remote ATtestation procedureS (RATS) Architecture", Work in Progress, Internet-Draft, draft-ietf-rats-architecture-22, 28 September 2022,

<https://datatracker.ietf.org/doc/html/draft-ietf-rats-architecture-22>.

[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-19, 19 December 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-rats-eat-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>.

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

[TLS-Param-Registry] IANA, "Transport Layer Security (TLS) Parameters", <https://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 5.2.

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 5.1. 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 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 10: 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