FIDO 2.0: Key Attestation Format

FIDO Alliance Proposed Standard 04 September 2015

This version:
https://fidoalliance.org/specs/fido-v2.0-ps-20150904/fido-key-attestation-v2.0-ps-20150904.html
Editor:
Rolf Lindemann, Nok Nok Labs
Contributor:
Michael B. Jones, Microsoft

The English version of this specification is the only normative version. Non-normative translations may also be available.

Abstract

The document defines generic data structures that cover the semantics of FIDO Authenticator attestation. The document also provides a profile of these structures when a TPM [TPM] acts as a crypto kernel. More profiles are expected to be added as the document evolves.

Attestation refers to the capability of a FIDO Authenticator to provide a cryptographic proof about its model to a remote relying party.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current FIDO Alliance publications and the latest revision of this technical report can be found in the FIDO Alliance specifications index at https://www.fidoalliance.org/specifications/.

This document was published by the FIDO Alliance as a Proposed Standard. If you wish to make comments regarding this document, please Contact Us. All comments are welcome.

Implementation of certain elements of this Specification may require licenses under third party intellectual property rights, including without limitation, patent rights. The FIDO Alliance, Inc. and its Members and any other contributors to the Specification are not, and

shall not be held, responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

THIS FIDO ALLIANCE SPECIFICATION IS PROVIDED “AS IS” AND WITHOUT ANY WARRANTY OF ANY KIND, INCLUDING, WITHOUT LIMITATION, ANY EXPRESS OR IMPLIED WARRANTY OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

This document has been reviewed by FIDO Aliance Members and is endorsed as a Proposed Standard. It is a stable document and may be used as reference material or cited from another document. FIDO Alliance's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment.

Table of Contents

1. Notation

1.1 Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words must, must not, required, should, should not, recommended, may, and optional in this specification are to be interpreted as described in [RFC2119].

1.2 Glossary

A glossary of terms used here is provided in a companion document.

The term **Base64url Encoding** refers to the base64 encoding using the URL- and filename-safe character set defined in Section 5 of [RFC4648], with all trailing '=' characters omitted (as permitted by Section 3.2) and without the inclusion of any line breaks, whitespace, or other additional characters. This is the same encoding as used by JSON Web Signature (JWS) [JWS].

2. Overview

2.1 Attestation Models

FIDO 2.0 specifies multiple attestation models:

Full Basic Attestation
: In the case of full basic attestation [UAFProtocol], the Authenticator's attestation private key is specific to an Authenticator model. That means that an Authenticator of the same model typically shares the same attestation private key.

  This model is also used for FIDO UAF 1.0 and FIDO U2F 1.0.

Surrogate Basic Attestation
: In the case of surrogate basic attestation [UAFProtocol], the Authenticator doesn't have any specific attestation key. Instead it uses the authentication key to (self-)sign the (surrogate) attestation message. Authenticators without meaningful protection measures for an attestation private key typically use this attestation model.

Privacy CA
: In this case, the authenticator owns an authenticator-specific (endorsement) key. This key is used to securely communicate with a trusted third party, the Privacy CA. The Authenticator can generate multiple attestation key pairs and asks the Privacy CA to issue an attestation certificate for it.

  Using this approach, the Authenticator can limit the exposure of the endorsement key (which is a global correlation handle) to Privacy CA(s). Attestation keys can be requested for each FIDO credential individually.

> NOTE
>
> This concept typically leads to multiple attestation certificates. The attestation

certificate requested most recently is called "active".

Direct Anonymous Attestation (DAA)
: In this case, the Authenticator receives DAA credentials from a single DAA-Issuer. These DAA credentials are used along with blinding to sign the attestation data. The concept of blinding avoids the DAA credentials being misused as global correlation handle.

  FIDO 2.0 supports DAA using elliptic curve cryptography and bilinear pairings, called ECDAA (see [FIDOEcdaaAlgorithm]) in this document.

Compliant FIDO Servers must support all attestation models. Authenticators can choose what attestation model to implement.

> NOTE
>
> Relying parties can always decide what attestation models are acceptable by policy.

2.2 Contextual Data

FIDO 2.0 attestation statements are bound to various contextual data. These data are observed, and added at different levels of the stack as a signature request passes from the server to the authenticator. In verifying a signature, the server checks these bindings against expected values.

These components can be divided into three layers:

1. The relying party (RP) consists of two subcomponents: an online service and a client-side application. The client-side app may, for example, be a web application running in a browser, or a native application that runs directly on the OS platform.
2. The client platform, which consists of the user's OS and device used to host the RP's client-side app. For web applications, the browser also belongs to this layer.
3. The authenticator itself, which provides key generation and key management and cryptographic signatures.

The goals of this design can be summarized as follows.

- The scheme for generating keys and attestation statements should accommodate cases where the link between the client platform and authenticator is very limited, in bandwidth and/or latency. Examples include Bluetooth Low Energy and Near-Field Communication.
- The data processed by the authenticator should be small and easy to interpret in low-level code. In particular, authenticators should not have to parse high-level encodings such as JSON.
- Both the client platform and the authenticator should have the flexibility to add contextual bindings, as needed.
- The design aims to reuse existing encoding formats as much as possible to aid adoption and implementation.

There are two kinds of contextual bindings: Those added by the RP or the client platform, referred to as **client data** (see [FIDOSignatureFormat]) and those added by the authenticator, referred to as the **attestation data**. The client data must be signed over, but an authenticator is otherwise not interested in its contents. More specifically, the authenticator cannot attest to the correctness of such data. To save bandwidth and processing requirements on the authenticator, the client platform hashes the client data and sends only the hash result to the authenticator. The authenticator attestation statement includes the combination of this hash, and its own attestation data.

2.3 Attestation Types

FIDO specifies pluggable attestation types, i.e., ways to serialize the data being attested to by the authenticator. The reason is to be able to support existing devices like TPMs and other platform-specific formats.

Each attestation type provides the ability to cryptographically attest to a public key, the authenticator model, and contextual data to a remote party.

3. Attestation Statement

When an attestation statement is required for an Authenticator, the client needs to ask the Authenticator to generate one. This section describes the attestation statement data structures. Attestation statements can also include some host and other Authenticator information.

The attestation statement consists of

1. the `header` object, containing the signing algorithm and additional information required to verify the attestation signature.
2. the `core` object, containing the attested data. This object is a container and can carry multiple, authenticator model specific, attestation rawData types (see section 3.4 Attestation Raw Data Types).
3. the `signature` object. This object contains the cryptographic signature computed over the `rawData` object. The structure of this object depends on the signature algorithm.

How the attestationStatement is returned from an external authenticator to the computer is described in the FIDO External Authenticator Protocol [FIDOEAP].

3.1 Client encoding of attestation statements

The client platform uses an authenticator generated attestation signature (`signature`) and the authenticator generated `rawData` object to construct the final attestation statement object (which will be returned to the RP). This attestation statement is encoded as:

WebIDL

```
interface AttestationStatement {
    readonly    attribute AttestationHeader header;
    readonly    attribute AttestationCore   core;
    readonly    attribute DOMString         signature;
};
```

3.1.1 Attributes

`header` of type *AttestationHeader*, readonly
: Attestation header, including algorithm, (optionally) the claimed AAGUID and (optionally) the attestation certificate chain.

`core` of type *AttestationCore*, readonly
: AttestationCore object. This object includes the attested `rawData` and its `type` and `version`.

`signature` of type DOMString, readonly
: The base64url-encoded attestation signature. The structure of this object depends on the signature algorithm specified in the header.

This attestation statement is delivered to the RP according to the Client API. It contains all the information that the RP's FIDO server requires to reconstruct the signature base string, as well as to decode and validate the bindings of both the client- and authenticator data.

3.2 AttestationCore

Data attested by the Authenticator and description of its structure. Different types of Authenticators might generate different object types (identified by `type` and `version`).

WebIDL

```
interface AttestationCore {
    readonly    attribute DOMString     type;
    readonly    attribute unsigned long version;
    readonly    attribute DOMString     rawData;
    readonly    attribute DOMString     clientData;
};
```

3.2.1 Attributes

`type` of type DOMString, readonly
: The type of the rawData object. This specification defines these attestation types: "tpm", "packed", and "android". Other attestation types may be defined in further versions of this specification.

`version` of type unsigned long, readonly
: The version number of the rawData object.

`rawData` of type DOMString, readonly
: The rawData object (for type "android"), or the base64url-encoded rawData object (for types "tpm" and "packed"), containing the attested public key and the `clientDataHash`.

`clientData` of type DOMString, readonly
: A base64url encoding of clientDataJSON [FIDOSignatureFormat]. The exact encoding must be preserved as the hash (clientDataHash) has been computed over it.

3.2.2 Client data

The client platform shall deliver, through the EAP API, the `clientDataHash` (see [FIDOSignatureFormat]) to the authenticator. The client platform must also preserve the exact `encodedClientData` string (see [FIDOSignatureFormat]), for embedding in a signature object sent back to the RP. This is necessary since multiple JSON encodings of the same client data are possible.

3.3 AttestationHeader

Additional data required to verify the attestation signature.

WebIDL

```
interface AttestationHeader {
    readonly    attribute DOMString   claimedAAGUID;
    readonly    attribute DOMString[] x5c;
    readonly    attribute DOMString   alg;
};
```

3.3.1 Attributes

`claimedAAGUID` of type DOMString, readonly
: The claimed Authenticator Attestation GUID (a version 4 GUID, see [RFC4122]). This value is used by the FIDO Server to lookup the trust anchor for verifying this AttestationCore object. If the verification succeeds, the AAGUID related to the trust anchor is trusted. This field must be present, if either no attestation certificates are used (e.g., DAA) or if the attestation certificate doesn't contain the AAGUID value in an extension.

`x5c` of type array of DOMString, readonly

Attestation Certificate and its certificate chain as described in [JWS] section 4.1.6.

`alg` of type DOMString, readonly
: The name of the algorithm used to generate the attestation signature according to [JWA].

  See section for the signature algorithms to be implemented by FIDO Servers.

3.4 Attestation Raw Data Types

Attestation Raw Data (`rawData`) is the to-be-signed object of the attestation statement. FIDO supports pluggable attestation data types. This allows support of TPM generated attestation data as well as support of other FIDO authenticators.

The contents of the attestation data must be controlled (i.e., generated or at least verified) by the authenticator itself.

3.4.1 Packed Attestation

Packed attestation is a FIDO optimized format of attestation data. It uses a very compact but still extensible encoding method. Encoding this format can even be implemented by authenticators with very limited resources (e.g., secure elements).

3.4.1.1 Attestation rawData (type="packed")

The attestation data encodes contextual bindings made by the authenticator itself. Unlike client data, the authenticator data has a compact but extensible encoding. This is desired since authenticators can be devices with limited capabilities and low power requirements, with much simpler software stacks than the client platform components.

For this type, only version="1" is defined at this time.

The field `rawData` is the **base64url encoding of the byte array**. The encoding of attestation data (for type "packed") is a byte array of 45 bytes + length of public key + length of KeyHandle + potentially more extensions. The first bytes before the extensions have a fixed layout as follows:

Length (in bytes)	Description
2	0xF1D0, fixed big-endian TAG to make sure this object won't be confused with other (non-FIDO) binary objects.
1	Flags (bit 0 is the least significant bit): • Bit 0: Test of User Presence (`TUP`) result. • Bits 1-6: Reserved for future use (`RFU`). • Bit 7: Extension data included (`ED`). Indicates whether the authenticator added extensions (see below).
4	Signature counter (`signCount`), 32-bit unsigned big-endian integer.
	Public key algorithm and encoding (16-bit big-endian value). Allowed values are: 1. 0x0100. This is raw ANSI X9.62 formatted Elliptic Curve public key [SEC1]. I.e., `[0x04, X (n bytes), Y (n bytes)]` . Where the byte `0x04` denotes the uncompressed point compression method and n denotes the key

2	length in bytes. 2. 0x0102. Raw encoded RSA PKCS1 or RSASSA-PSS public key [RFC3447]. In the case of RSASSA-PSS, the default parameters according to [RFC4055] must be assumed, i.e., ◦ Mask Generation Algorithm MGF1 with SHA256 ◦ Salt Length of 32 bytes, i.e., the length of a SHA256 hash value. ◦ Trailer Field value of 1, which represents the trailer field with hexadecimal value `0xBC`. That is, `[modulus (256 bytes), e (m-n bytes)]`. Where `m` is the total length of the field. This total length should be taken from the object containing this key
2	Byte length m of following public key bytes (16 bit value with most significant byte first).
(length)	The public key (m bytes) according to the encoding denoted before.
2	Byte length l of KeyHandle
(length)	KeyHandle (l bytes)
2	Byte length n of clientDataHash
n	clientDataHash (see section 3.2.2 Client data). This is the hash of `clientData`. The hash algorithm itself is stored in the `clientData` object [FIDOSignatureFormat].
As defined by the extension map	Extension-defined authenticator data. This is a CBOR [RFC7049] map with extension identifiers as keys, and extension authenticator data values as values. See [FIDOSignatureFormat], section "Signature extensions" for a description of the extension mechanism. See section 3.4.1.2 Extensions for Packed Attestation rawData for pre-defined extensions.

The `TUP` flag shall be set if and only if the authenticator detected a user through an authenticator-specific gesture. The `RFU` bits in the flags byte shall be cleared (i.e., zeroed).

If the authenticator does not wish to add extensions, itmust clear the `ED` flag in the third byte.

3.4.1.2 Extensions for Packed Attestation rawData

3.4.1.2.1 AAGUID Extension

Extension identifier
`fido.aaguid`
Client argument
N/A
Client processing
N/A
Authenticator argument
N/A
Authenticator processing

This extension is added automatically by the authenticator. This extension can be added to attestation statements and signatures.

Authenticator data

A 128-bit Authenticator Attestation GUID encoded as a CBOR text string (major type 3).

This AAGUID is used to identify the Authenticator model (Authenticator Attestation GUID).

> NOTE
>
> The authenticator model (identified by the AAGUID) can be derived from (a) here, or (b) from the attestation certificate (if we have an authenticator specific or authenticator model specific attestation certificate), or (c) or from the claimed AAGUID in the client encoded attestation statement (if we have one attestation root certificate per authenticator model).
>
> In the case of DAA there is no need for an X.509 attestation certificate hierarchy. Instead the trust anchor being known to the RP is the DAA root key (i.e. ECPoint2 X, Y). This root key must be dedicated to a single authenticator model.

3.4.1.2.2 SupportedExtensions Extension

Extension identifier

`fido.exts`

Client argument

N/A

Client processing

N/A

Authenticator argument

N/A

Authenticator processing

This extension is added automatically by the authenticator. This extension can be added to attestation statements.

Authenticator data

The SupportedExtension extension is a list (CBOR array) of extension identifiers encoded as UTF-8 Strings.

3.4.1.2.3 User Verification Index (UVI) Extension

Extension identifier

`fido.uvi`

Client argument

N/A

Client processing

N/A

Authenticator argument

N/A

Authenticator processing

This extension is added automatically by the authenticator. This extension can be added to attestation statements and signatures.

Authenticator data

The user verification index (UVI) is a value uniquely identifying a user verification data record. The UVI is encoded as CBOR byte string (type 0x58).

Each UVI value must be specific to the related key (in order to provide unlinkability). It also must contain sufficient entropy that makes guessing impractical. UVI values

must not be reused by the Authenticator (for other biometric data or users).

The UVI data can be used by FIDO Servers to understand whether an authentication was authorized by the exact same biometric data as the initial key generation. This allows the detection and prevention of "friendly fraud".

As an example, the UVI could be computed as SHA256(KeyID | SHA256(rawUVI)), where the rawUVI reflects (a) the biometric reference data, (b) the related OS level user ID and (c) an identifier which changes whenever a factory reset is performed for the device, e.g. rawUVI = biometricReferenceData | OSLevelUserID | FactoryResetCounter.

FIDO Servers supporting UVI extensions must support a length of up to 32 bytes for the UVI value.

Example for rawData containing one UVI extension

```
F1 D0                          -- This is a FIDO packed rawData object
81                             -- TUP and ED set
00 00 00 01                    -- (initial) signature counter
...                            -- all public key alg etc.
A1                             -- extension: CBOR map of one element
  68                           -- Key 1: CBOR text string of 8 bytes
    66 69 64 6F 2E 75 76 69    -- "fido.uvi" UTF-8 string
  58 20                        -- Value 1: CBOR byte string with 0x20 bytes
  00 43 B8 E3 BE 27 95 8C      -- the UVI value itself
  28 D5 74 BF 46 8A 85 CF
  46 9A 14 F0 E5 16 69 31
  DA 4B CF FF C1 BB 11 32
  82
```

3.4.1.3 Signature

The `signature` is computed over the base64url-decoded `rawData` field.

The following algorithms must be implemented by FIDO Servers:

1. "ES256" [RFC7518]
2. "RS256" [RFC7518]
3. "PS256" [RFC7518]
4. "ED256" [FIDOEcdaaAlgorithm]

Authenticators can choose which algorithm to implement.

3.4.1.4 Attestation statement certificate requirements

> NOTE
>
> In the case of DAA attestation [FIDOEcdaaAlgorithm] no attestation certificate is used.

The attestation certificate must have the following fields/extensions:

- Version must be set to 3.
- Subject field must be set to:

  Subject-C
  : Country where the Authenticator vendor is incorporated

  Subject-O
  : Legal name of the Authenticator vendor

  Subject-OU

Authenticator Attestation

Subject-CN

No stipulation.

- If the related attestation root certificate is used for multiple authenticator models, the following extension must be present:

  Extension OID `1 3 6 1 4 1 45724 1 1 4` (id-fido-gen-ce-aaguid) containing the AAGUID as value.

- The Basic Constraints extension must have the cA component set to false
- An Authority Information Access (AIA) extension with entry `id-ad-ocsp` and a CRL Distribution Point extension [RFC5280] are both optional as the status of attestation certificates is available through the FIDO Metadata Service [FIDOMetadataService].

3.4.2 TPM Attestation

3.4.2.1 Attestation rawData (type="tpm")

The value of `rawData` is the **base64url encoding of a binary object**. This binary object is either a TPM_CERTIFY_INFO or a TPM_CERTIFY_INFO2 structure [TPMv1-2-Part2] sections 11.1 and 11.2, if `attestationStatement.core.version` equals 1. Else, if `attestationStatement.core.version` equals 2, it must be the base64url encoding of a TPMS_ATTEST structure as defined in [TPMv2-Part2] sections 10.12.9.

The field "extraData" (in the case of TPMS_ATTEST) or the field "data" (in the case of TPM_CERTIFY_INFO or TPM_CERTIFY_INFO2) must contain the `clientDataHash` (see [FIDOSignatureFormat]).

3.4.2.2 Signature

If `attestationStatement.core.version` equals 1, (i.e., for TPM 1.2), RSASSA-PKCS1-v1_5 signature algorithm (section 8.2 of [RFC3447]) can be used by FIDO Authenticators (i.e. `attestationStatement.header.alg`="RS256").

If `attestationStatement.core.version` equals 2, the following algorithms can be used by FIDO Authenticators:

1. TPM_ALG_RSASSA (0x14). This is the same algorithm RSASSA-PKCS1-v1_5 as for version 1 but for use with TPMv2. `attestationStatement.header.alg`="RS256".
2. TPM_ALG_RSAPSS (0x16); `attestationStatement.header.alg`="PS256".
3. TPM_ALG_ECDSA (0x18); `attestationStatement.header.alg`="ES256".
4. TPM_ALG_ECDAA (0x1A); `attestationStatement.header.alg`="ED256".
5. TPM_ALG_SM2 (0x1B); `attestationStatement.header.alg`="SM256".

The `signature` is computed over the base64url-decoded `rawData` field

See section for the signature algorithms to be implemented by FIDO Servers.

3.4.2.3 TPM attestation statement certificate requirements

TPM attestation certificate must have the following fields/extensions:

- Version must be set to 3.
- Subject field must be set to empty.
- The Subject Alternative Name extension must be set as defined in [TPMv2-EK-Profile] section 3.2.9 if "version" equals 2 and [TPMv1-2-Credential-Profiles] section 3.2.9 if "version" equals 1.

- The Extended Key Usage extension must contain the "joint-iso-itu-t(2) internationalorganizations(23) 133 tcg-kp(8) tcg-kp-AIKCertificate(3)" OID.
- The Basic Constraints extension must have the CA component set to false
- An Authority Information Access (AIA) extension with entry `id-ad-ocsp` and a CRL Distribution Point extension [RFC5280] are both optional as the status of attestation certificates is available through the FIDO Metadata Service [FIDOMetadataService].

3.4.3 Android Attestation

When the Authenticator in question is a platform-provided Authenticator on the Android platform, the attestation statement is based on the SafetyNet API.

Android attestation statement must always be used in conjunction with the more specific `AndroidAttestationClientData` (see section 3.4.3.4 AndroidAttestationClientData) in order to let the RP App store the public key in the attestation object.

3.4.3.1 Attestation rawData (type="android")

Android SafetyNet returns a JWS [JWS] object (see SafetyNet online documentation) in Compact Serialization. A JWS in Compact Serialization consists of three segments (each a base64url-encoded string) separated by a dot ("."). The `rawData` object is the concatenation of:

1. the first segment (a base64url encoding of the UTF-8 encoded JWS Protected Header)
2. a dot "."
3. the second segment (a base64url encoding of the UTF-8 encoded JWS Payload).

In contrast to the "packed" and "tpm" attestation types, for the "android" attestation type, the `rawData` field and the `rawData` object are the same string. (In the "packed" and "tpm" attestation types the `rawData` field is the base64url-encoding of the `rawData` object.)

3.4.3.2 Signature

The signature is directly computed over the `rawData` field, as defined above (see [JWS] for more details). The third segment of the JWS returned by SafetyNet is the base64url encoding of this signature, and also becomes the `AttestationStatement.signature`.

3.4.3.3 Converting SafetyNet response to attestationStatement

The Authenticator and/or platform should use the steps outlined below to create an attestationStatement from an Android SafetyNet response. It may use a different algorithm, as long as the results are the same.

1. create clientDataJSON with type AndroidAttestationClientData (see below) and compute clientData as base64url-encoded clientDataJSON.
2. provide the clientDataHash computed as the hash value of clientData as Nonce value when requesting the SafetyNet attestation.
3. take SafetyNet response **snr**. This is a JWS object ([JWS]).
4. extract the base64url-encoded Protected Header **hdr** from *snr* (see [JWS])
5. extract the base64url-encoded payload **p** from *snr*
6. extract the base64url-encoded signature **s** from *snr*
7. set `AttestationStatement.core.rawData` = *hdr* | "." | *p*
8. set `AttestationStatement.signature` = *s*
9. base64url-decode *hdr* into **hdr-d**
10. set `AttestationStatement.header.alg` = *hdr-d*.alg

11. if *hdr-d*.x5c is present, then set `AttestationStatement.header.x5c` = *hdr-d*.x5c
12. if *hdr-d*.x5u is present, then resolve the URL and add the retrieved certificate chain to `AttestationStatement.header.x5c`
13. set `AttestationStatement.core.type` = "android"
14. set `AttestationStatement.core.version` to the version number of Google Play Services responsible for providing the SafetyNet API

3.4.3.4 AndroidAttestationClientData

The ClientData dictionary is extended in the following way:

WebIDL

```
dictionary AndroidAttestationClientData : ClientData {
    JsonWebKey              publicKey;
    boolean                 isInsideSecureHardware;
    DOMString               userAuthentication;
    optional unsigned long  userAuthenticationValidityDurationSeconds;
};
```

3.4.3.4.1 Dictionary `AndroidAttestationClientData` Members

`publicKey` of type JsonWebKey
: The public key generated by the Authenticator, as a JsonWebKey object (see [WebCrypto]).

`isInsideSecureHardware` of type boolean
: `true` if the key resides inside secure hardware (e.g., Trusted Execution Environment (TEE) or Secure Element (SE)).

`userAuthentication` of type DOMString
: One of "none", "keyguard", or "fingerprint". *none* means that the user has not enrolled a fingerprint, or set up a secure lock screen, and that therefore the key has not been linked to user authentication. *keyguard* means that the generated key only be used after the user unlocks a secure lock screen. *fingerprint* means that each operation involving the generated key must be individually authorized by the user by presenting a fingerprint.

`userAuthenticationValidityDurationSeconds` of type optional unsigned long
: If the `userAuthentication` is set to "keyguard", then this parameter specifies the duration of time (seconds) for which this key is authorized to be used after the user is successfully authenticated.

3.4.3.4.2 Verifying AndroidClientData specific contextual bindings

A relying party shall verify the clientData contextual bindings (see step 4 in 3.5 Verifying an Attestation Statement as follows:

- Check that `AndroidAttestationClientData.challenge` equals the attestationChallenge that was passed into the `makeCredential` call [FIDOWebApi].
- Check that the `facet` and `tokenBinding` parameters in the `AndroidAttestationClientData` match the RP App.
- Check that `AndroidAttestationClientData.publicKey` is the same key as the one returned in the `FIDOCredentialInfo` by the `makeCredential` call.
- Check that the hash of the clientDataJSON matches the `nonce` attribute in the payload of the `safetynetResponse` JWS.

- Check that the `ctsProfileMatch` attribute in the payload of the `safetynetResponse` is true.
- Check that the `apkPackageName` attribute in the payload of the `safetynetResponse` matches package name of the application calling SafetyNet API.
- Check that the `apkDigestSha256` attribute in the payload of the `safetynetResponse` matches the package hash of the application calling SafetyNet API.
- Check that the `apkCertificateDigestSha256` attribute in the payload of the `safetynetResponse` matches the hash of the signing certificate of the application calling SafetyNet API.

3.5 Verifying an Attestation Statement

This section outlines the recommended algorithm for verifying an attestation statement, independent of attestation type.

Upon receiving an attestation statement, the relying party shall:

1. Verify that the attestation statement is properly formatted
2. If `attestationSignature.alg` is not ECDAA (e.g., not "ED256" and not "ED512"):
    1. Lookup the attestation root certificate from a trusted source. The FIDO Metadata Service [FIDOMetadataService] provides an easy way to access such information. The `header.claimedAAGUID` can be used for this lookup.
    2. Verify that the attestation certificate chain is valid and chains up to a trusted root (following [RFC5280]).
    3. Verify that the attestation certificate has the right Extended Key Usage (EKU) based on the FIDO Authenticator type (as denoted by the `header.type` member). In case of a type="tpm", this EKU shall be OID "2.23.133.8.3".
    4. If the attestation type is "android", verify that the attestation certificate is issued to the hostname "attest.android.com" (see SafetyNet online documentation).
    5. Verify that all issuing CA certificates in the chain are valid and not revoked.
    6. Verify the `signature` on `core.rawData` using the attestation certificate public key and algorithm as identified by `header.alg`.
    7. Verify `core.rawData` syntax and that it doesn't contradict the signing algorithm specified in `header.alg`.
    8. If the attestation certificate contains an extension with OID `1 3 6 1 4 1 45724 1 1 4` (id-fido-gen-ce-aaguid) verify that the value of this extension matches `header.claimedAAGUID`. This identifies the Authenticator model.
    9. If such extension doesn't exist, the attestation root certificate is dedicated to a single Authenticator model.
3. If `attestationSignature.alg` is ECDAA (e.g., "ED256", "ED512"):
    1. Lookup the DAA root key from a trusted source. The FIDO Metadata Service [FIDOMetadataService] provides an easy way to access such information. The `header.claimedAAGUID` can be used for this lookup.
    2. Perform DAA-Verify on `signature` for `core.rawData` (see [FIDOEcdaaAlgorithm]).
    3. Verify `core.rawData` syntax and that it doesn't contradict the signing algorithm specified in `header.alg`.
    4. The DAA root key is dedicated to a single Authenticator model.
4. Verify the contextual bindings (e.g., channel binding) from the clientData (see Section 3.2.2 Client data).
5. Verify that user verification method and other authenticator characteristics related to this authenticator model, match the relying party policy. The FIDO Metadata Service [FIDOMetadataService] provides an easy way to access the authenticator characteristics.

The relying party may take any of the below actions when verification of an attestation statement fails:

- Reject the request, such as a registration request, associated with the attestation statement.
- Accept the request associated with the attestation statement but treat the attested FIDO Credential as one with surrogate basic attestation (see 2.1 Attestation Models), if policy allows it. If doing so, there is no cryptographic proof that the FIDO Credential has been generated by a particular Authenticator model. See [FIDOSecRef] and [UAFProtocol] for more details on the relevance on attestation.

Verification of attestation statements requires that the relying party trusts the root of the attestation certificate chain. Also, the relying party must have access to certificate status information for the intermediate CA certificates. The relying party must also be able to build the attestation certificate chain if the client didn't provide this chain in the attestation information.

4. Security Considerations

4.1 Privacy

Attestation keys may be used to track users or link various online identities of the same user together. This may be mitigated in several ways, including:

- A FIDO Authenticator manufacturer may choose to ship all of their devices with the same (or a fixed number of) attestation key(s) (called Full Basic Attestation). This will anonymize the user at the risk of not being able to revoke a particular attestation key should its FIDO Authenticator be compromised.
- A FIDO Authenticator may be capable of dynamically generating different attestation keys (and requesting related certificates) per origin (following the Privacy CA model). For example, a FIDO Authenticator can ship with a master attestation key (and certificate), and combined with a cloud operated privacy CA, can dynamically generate per origin attestation keys and attestation certificates.
- A FIDO Authenticator can implement direct anonymous attestation (see [FIDOEcdaaAlgorithm]). Using this scheme the authenticator generates a blinded attestation signature. This allows the relying party to verify the signature using the DAA root key, but the attestation signature doesn't serve as a global correlation handle.

4.2 Attestation Certificate and Attestation Certificate CA Compromise

When an intermediate CA or a root CA used for issuing attestation certificates is compromised, FIDO Authenticator attestation keys are still safe although their certificates can no longer be trusted. A FIDO Authenticator manufacturer that has recorded the public attestation keys for their devices can issue new attestation certificates for these keys from a new intermediate CA or from a new root CA. If the root CA changes, the relying parties must update their trusted root certificates accordingly.

A FIDO Authenticator attestation certificate must be revoked by the issuing CA if its key has been compromised. A FIDO Authenticator manufacturer may need to ship a firmware update and inject new attestation keys and certificates into already manufactured FIDO Authenticators, if the exposure was due to a firmware flaw. (The process by which this happens is out of scope for this specification.) No further valid attestation statements can be made by the affected FIDO Authenticators unless the FIDO Authenticator manufacturer has this capability.

If attestation certificate validation fails due to a revoked intermediate attestation CA certificate, and relying party policy requires rejecting the registration/authentication request in these situations, then it is recommended that the relying party also un-registers (or marks as "surrogate attestation" (see 2.1 Attestation Models), policy permitting) FIDO credentials that were registered post the CA compromise date using an attestation certificate chaining up to the same intermediate CA. It is thus recommended that relying parties remember intermediate attestation CA certificates during Authenticator registration in order to un-register related FIDO Credentials if the registration was performed after revocation of such certificates.

If a DAA attestation key has been compromised, it can be added to the RogueList (i.e., the list of revoked authenticators) maintained by the related DAA-Issuer. The relying party should verify whether an authenticator belongs to the RogueList when performing DAA-Verify. The FIDO Metadata Service [FIDOMetadataService] provides an easy way to access such information.

4.3 Attestation Certificate Hierarchy

A 3 tier hierarchy for attestation certificates is recommended (i.e., Attestation Root, Attestation Issuing CA, Attestation Certificate). It is also recommended that for each FIDO Authenticator device line (i.e., model), a separate issuing CA is used to help facilitate isolating problems with a specific version of a device.

If the attestation root certificate is *not* dedicated to a single FIDO Authenticator device line (i.e., AAGUID), the AAGUID must be specified either in the attestation certificate itself or as an extension in the `core.rawData`.

A. References

A.1 Normative references

[FIDOEcdaaAlgorithm]
R. Lindemann, A. Edgington, R. Urian, *FIDO ECDAA Algorithm*. FIDO Alliance Proposed Standard. URLs:
HTML: fido-ecdaa-algorithm-ps-20141208.html
PDF: fido-ecdaa-algorithm-ps-20141208.pdf

[FIDOSignatureFormat]
FIDO 2.0: Signature format. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20150904/fido-signature-format.html

[JWA]
M. Jones. *JSON Web Algorithms (JWA)*. May 2015. RFC. URL: https://tools.ietf.org/html/rfc7518

[JWS]
M. Jones; J. Bradley; N. Sakimura. *JSON Web Signature (JWS)*. May 2015. RFC. URL: https://tools.ietf.org/html/rfc7515

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels* March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC7518]
M. Jones. *JSON Web Algorithms (JWA)*. May 2015. Proposed Standard. URL: https://tools.ietf.org/html/rfc7518

[SEC1]
Standards for Efficient Cryptography Group (SECG), *SEC1: Elliptic Curve Cryptography*, Version 2.0, September 2000.

[WebCrypto]
R. Sleevi; M. Watson. *Web Cryptography API*. 11 December 2014. W3C Candidate Recommendation. URL: http://www.w3.org/TR/WebCryptoAPI/

A.2 Informative references

[FIDOEAP]
Reference not found.

[FIDOMetadataService]
R. Lindemann, B. Hill, D. Baghdasaryan, *FIDO Metadata Service v1.0*. FIDO Alliance Proposed Standard. URLs:
HTML: fido-metadata-service-v1.1-id-20150902.html
PDF: fido-metadata-service-v1.1-id-20150902.pdf

[FIDOSecRef]
R. Lindemann, D. Baghdasaryan, B. Hill, *FIDO Security Reference*. FIDO Alliance Proposed Standard. URLs:
HTML: fido-security-ref-v1.1-id-20150902.html
PDF: fido-security-ref-v1.1-id-20150902.pdf

[FIDOWebApi]
FIDO 2.0: Web API for accessing FIDO 2.0 credentials URL:

https://fidoalliance.org/specs/fido-v2.0-ps-20150904/fido-web-api.html

[RFC3447]
J. Jonsson; B. Kaliski. *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1*. February 2003. Informational. URL: https://tools.ietf.org/html/rfc3447

[RFC4055]
J. Schaad; B. Kaliski; R. Housley. *Additional Algorithms and Identifiers for RSA Cryptography for use in the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. June 2005. Proposed Standard. URL: https://tools.ietf.org/html/rfc4055

[RFC4122]
P. Leach; M. Mealling; R. Salz. *A Universally Unique IDentifier (UUID) URN Namespace*. July 2005. Proposed Standard. URL:https://tools.ietf.org/html/rfc4122

[RFC4648]
S. Josefsson, *The Base16, Base32, and Base64 Data Encodings (RFC 4648)*, IETF, October 2006, URL: http://www.ietf.org/rfc/rfc4648.txt

[RFC5280]
D. Cooper, S. Santesson, s. Farrell, S.Boeyen, R. Housley, W. Polk;*Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, IETF, May 2008, URL: http://www.ietf.org/rfc/rfc5280.txt

[TPM]
TPM Main Specification Trusted Computing Group. Accessed March 2014. URL: http://www.trustedcomputinggroup.org/resources/tpm_main_specification

[TPMv1-2-Credential-Profiles]
Trusted Computing Group, *TPM 1.2 Credential Profiles*, URL: http://www.trustedcomputinggroup.org/files/static_page_files/A55529C5-1A4B-B294-D0A5A400E1EDE13A/Credential_Profiles_V1.2_Level2_Revision8.pdf

[TPMv1-2-Part2]
Trusted Computing Group, *TPM 1.2 Part 2: Structures*, URL: http://www.trustedcomputinggroup.org/files/static_page_files/E55A303C-1A4B-B294-D066E66A82DAE27D/TPM%20Main-Part%202%20TPM%20Structures_v1.2_rev116_01032011.pdf

[TPMv2-EK-Profile]
Trusted Computing Group, *TCG EK Credential Profile*, URL: http://www.trustedcomputinggroup.org/files/static_page_files/DCD56924-1A4B-B294-D0CEF64E80CEE01E/Credential_Profile_EK_V2.0_R12_PublicReview.pdf

[TPMv2-Part2]
Trusted Computing Group, *Trusted Platform Module Library, Part 2: Structures*, URL: http://www.trustedcomputinggroup.org/files/static_page_files/8C583202-1A4B-B294-D0469592DB10A6CD/TPM%20Rev%202.0%20Part%202%20-%20Structures%2001.16.pdf

[UAFProtocol]
R. Lindemann, D. Baghdasaryan, E. Tiffany, D. Balfanz, B. Hill, J. Hodges,*FIDO UAF Protocol Specification v1.0*. FIDO Alliance Proposed Standard. URLs:
HTML: fido-uaf-protocol-v1.1-id-20150902.html
PDF: fido-uaf-protocol-v1.1-id-20150902.pdf