Table of Contents

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

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 I 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 I OSLevelUserID I 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

FIDO 2.0: Signature format

FIDO Alliance Proposed Standard 04 September 2015

This version:
https://fidoalliance.org/specs/fido-v2.0-ps-20150904/fido-signature-format-v2.0-ps-20150904.html
Editors:
Arnar Birgisson, Google
Michael B. Jones, Microsoft
Alexei Czeskis, Google
Contributor:
Rolf Lindemann, Nok Nok Labs

Abstract

A FIDO 2.0 signature proves possession of a private key of a FIDO 2.0 credential and asserts contextual information about the client and authenticator that generated it. This document describes the data structures representing these assertions, how they are serialized to byte streams for signing with an authenticator, and the representation of the resulting signature and its associated data.

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.

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. 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].

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].

1.1 Dependencies

This specification relies on several other underlying specifications.

HTML5
: The concept of **origin** and the **Window** interface are defined in [HTML5].

Web IDL
: Many of the interface definitions and all of the IDL in this specification depend on [WebIDL-ED]. This updated version of the Web IDL standard adds support for **Promises**, which are now the preferred mechanism for asynchronous interaction in all new web APIs.

DOM
: **DOMException** and the DOMException values used in this specification are defined in [DOM4].

FIDO External Authenticator Protocol
: This specification references methods for the client to communicate with FIDO 2.0 authenticators. These methods are specified in [FIDOEAP].

Web Cryptography API
: The **AlgorithmIdentifier** type and the method for normalizing an algorithm are defined in [WebCrypto].

2. Overview

FIDO 2.0 signatures 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.

The components of a system using FIDO 2.0 can be divided into three layers:

1. The relying party (RP), which uses the FIDO 2.0 services. The relying party 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 management and cryptographic signatures.

When the RP client-side application is a web-application, the interface between 1 and 2 is the FIDO 2.0 Web API [FIDOWebApi], but is platform specific for native applications. In cases where the authenticator is not tightly integrated with the platform, the interface between 2 and 3 is the FIDO External Authenticator Protocol [FIDOEAP]. This document defines the common signature format shared by all layers. This includes how the different contextual bindings are encoded, signed over, and delivered to the RP.

The goals of this design can be summarized as follows.

- The scheme for generating signatures 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 as much as possible of existing encoding formats in order to aid adoption and implementation.

The contextual bindings are divided in two: Those added by the RP or the client platform, referred to as **client data**; and those added by the authenticator, referred to as the **authenticator data**. The client data must be signed over, but an authenticator is otherwise not interested in its contents. To save bandwidth and processing requirements on the authenticator, the client platform hashes the client data and sends only the result to the authenticator. The authenticator signs over the combination of this hash, and its own authenticator data.

3. Client data

The client data represents the contextual bindings of both the RP and the client platform. It is a key-value mapping with string-valued keys. Values may be any type that has a valid encoding in JSON. It must contain at least the following key-value pairs.

WebIDL

```
dictionary ClientData {
    DOMString       challenge;
    DOMString       facet;
    JsonWebKey      tokenBinding;
    optional object extensions;
    DOMString       hashAlg;
};
```

3.1 Dictionary ClientData Members

`challenge` of type DOMString
: A base64url-encoded challenge provided by the RP.

`facet` of type DOMString
: A string value describing the RP identifier facet [FIDOPlatformApiReqs]. When the RP client-side app is a website, this is its fully qualified web origin, using the syntax defined by [RFC6454]. When the client-side app is a native application, this string is a platform specific identifier.

`tokenBinding` of type JsonWebKey
: A JsonWebKey object [JWK] describing the public key that this client uses for the Token Binding protocol when communicating with the Relying Party. This can be omitted if no Token Binding has been negotiated between the client and the Relying Party.

`extensions` of type optional object
: An object with extension-provided authenticator data. Signature extensions are detailed in Section 7. FIDO Extensions.

`hashAlg` of type DOMString
: The hash algorithm used to compute clientDataHash (see 5. Generating a signature). Use "S256" for SHA-256, "S384" for SHA384, "S512" for SHA512, and "SM3" for SM3 (see 8. IANA Considerations).

The client data may contain additional properties.

Before making a request to an authenticator, the client platform layer shall perform the following steps.

1. Let `clientDataJSON` be the UTF-8 encoded JSON serialization [RFC7159] of `clientData`.

2. Let `clientDataHash` be the hash (computed using `hashAlg`) of `clientDataJSON`, as an array.

The `clientDataHash` value is incorporated into a signature by a FIDO authenticator (see 5. Generating a signature). It is delivered to integrated authenticators in platform specific manners, and to external authenticators as a part of a signature request as specified by the External Authenticator Protocol [FIDOEAP]. The client platform should also preserve the exact `encodedClientData` string used to create it, for embedding in a signature object sent back to the RP (see 5. Generating a signature). This is necessary since multiple JSON encodings of the same client data are possible.

The hash algorithm `hashAlg` used to compute `clientDataHash` is included in the `clientData` object. This way it is available to the RP and it is also hashed over when computing `clientDataHash` and hence anchored in the signature itself.

4. Authenticator data

The authenticator data encodes contextual bindings made by the authenticator itself. 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.

The encoding of authenticator data is a byte array `authenticatorData` of 5 bytes or more, as follows.

Byte index	Description
0	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 if the authenticator data has extensions.
1-4	Signature counter (`signCount`), 32-bit unsigned big-endian integer.
5-	Extension-defined authenticator data. This is a CBOR [RFC7049] map with extension identifiers as keys, and extension authenticator data values as values. See 7. FIDO Extensions for details.

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 set to zero.

If the authenticator does not include any extension data, it must set the `ED` flag in the first byte to zero, and to one if extension data is included.

Fig. 1 `authenticatorData` layout. shows a visual representation of the authenticator data structure.

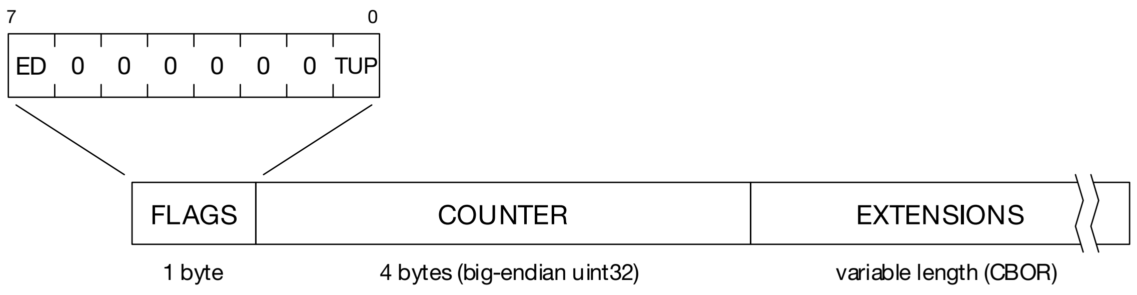

Fig. 1 `authenticatorData` layout.

NOTE

The `signatureData` describes its own length: If the ED flag is not set, it is always 5 bytes long. If the ED flag is set, it is 5 bytes plus the CBOR map that follows.

5. Generating a signature

A raw cryptographic signature must assert the integrity of both the client data and the authenticator data. Thus, an authenticator shall compute a signature over the concatenation of the `authenticatorData` and the `clientDataHash`.

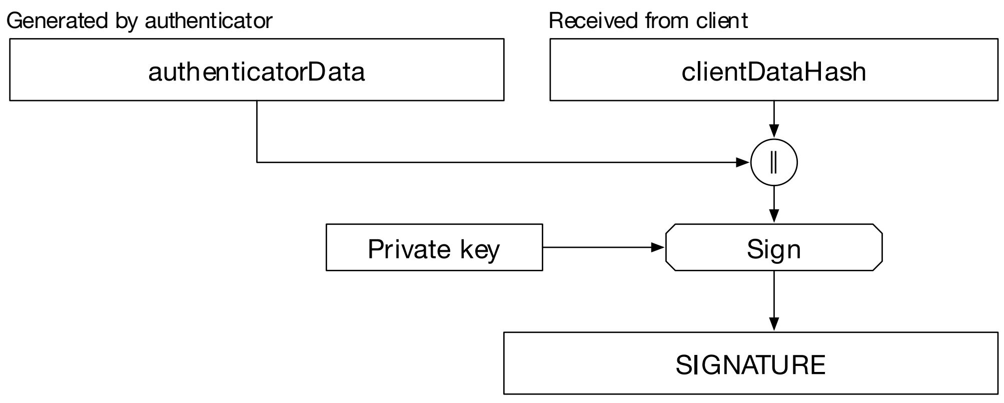

Fig. 2 Generating a signature on the authenticator.

NOTE

A simple, undelimited concatenation, is safe to use here because the `authenticatorData` describes its own length. The `clientDataHash` (which potentially has a variable length) is always the last element.

The authenticator must return both the `authenticatorData` and the raw signature back to the client.

6. Client encoding of assertions

The client platform uses an authenticator assertion to construct the final **FIDO assertion** object returned to the RP as follows.

WebIDL

```
interface FIDOAssertion {
                 attribute Credential  credential;
                 attribute DOMString   clientData;
                 attribute DOMString   authenticatorData;
                 attribute DOMString   signature;
};
```

6.1 Attributes

`credential` of type Credential,
An object representing which credential was used to generate an assertion.

`clientData` of type DOMString,
A base64url encoding of `clientDataJSON`. (See 3. Client data)

`authenticatorData` of type DOMString,
A base64url encoding of `authenticatorData`. (See 4. Authenticator data)

`signature` of type DOMString,
A base64url encoding of the raw signature returned from the authenticator. (See 5. Generating a signature)

This assertion is delivered to the RP in either a platform specific manner, or in the case of web applications, according to the FIDO Web API [FIDOWebApi]. 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.

7. FIDO Extensions

The mechanism for generating FIDO 2.0 credentials, as well as requesting and generating FIDO 2.0 assertions, as defined in [FIDOWebApi] and in this document, can be extended to suit particular use cases. Each case is addressed by defining a *registration extension* and/or a *signature extension*. Extensions can define additions to the following steps and data:

- `makeCredential` request parameters (see [FIDOWebApi]) for registration extension.
- `getAssertion` request parameters (see [FIDOWebApi]) for signature extensions.
- Client processing, and the `clientData` structure, for registration extensions and signature extensions.
- Authenticator processing, and the `authenticatorData` structure, for signature extensions.

When requesting an assertion for a FIDO 2.0 credential, an RP can list a set of extensions to be used, if they are supported by the client and/or the authenticator. It sends the request parameters for each extension in the `getAssertion` call (for signature extensions) or `makeCredential` call (for registration extensions) to the client platform. The client platform performs additional processing for each extension that it supports, and augments `clientData` as required by the extension. For extensions that the client platform does not support, it passes the request parameters on to the authenticator when possible (criteria defined below). This allows one to define extensions that affect the authenticator only.

Similarly, the authenticator performs additional processing for the extensions that it supports, and augments `authenticatorData` as specified by the extension.

Extensions that are not supported are ignored.

7.1 Extension identifiers

Extensions are identified by a string, chosen by the extension author. Extension identifiers should aim to be globally unique, e.g. by using reverse domain-name of the defining entity such as `com.example.fido.myextension`.

Extensions that may exist in multiple versions should take care to include a version in their identifier. In effect, different versions are thus treated as different extensions.

Standard extensions defined by FIDO in this document use a fixed prefix of `fido.` for the extension identifiers. This prefix should not be used for 3rd party extensions.

7.2 Defining extensions

A definition of an extension must specify, at minimum, an extension identifier and an extension client argument sent via the `getAssertion` or `makeCredential` call (see below). Additionally, extensions may specify additional values in `clientData`, `authenticatorData` (in the case of signature extensions), or both.

> NOTE
>
> An extension that does not define additions to `clientData` nor `authenticatorData` is possible, but should be avoided. In such cases, the relying party would have no indication if the extension was supported or processed by the client and/or authenticator.

7.2.1 Extending request parameters

An extension defines two request arguments. The **client argument** is passed from the RP to the client in the `getAssertion` or `makeCredential` call, while the **authenticator argument** is passed from the client to the authenticator during the processing of these calls, either natively or through the external authenticator protocol [FIDOEAP].

Extension definitions must specify the valid values for their client argument. Clients are free to ignore extensions with an invalid client argument. Specifying an authenticator argument is optional, since some extensions may only affect client processing.

An RP simultaneously requests the use of an extension and sets its client argument by including an entry in the `extensions` dictionary parameter to the `getAssertion` or `makeCredential` call. The entry key must be the extension identifier, and the value must be the client argument.

EXAMPLE 1

```
var assertionPromise = credentials.getAssertion(..., /* extensions */ {
  "com.example.fido.foobar": 42
});
```

Extensions that affect the behavior of the client platform can define their argument to be any set of values that can be encoded in JSON. Such an extension will in general (but not always) specify additional values to the `clientData` structure (see below). It may also specify an authenticator argument that platforms implementing the extension are expected to send to the authenticator. The authenticator argument should be a byte string.

> NOTE
>
> Extensions should aim to define authenticator arguments that are as small as possible. Some authenticators communicate over low-bandwidth links such as Bluetooth Low-Energy or NFC.

> NOTE
>
> Extensions that do not need to pass any particular argument value, must still define a client argument. It is recommended that the argument be defined as the constant value `true` in this case.

For extensions that specify additional authenticator processing only, it is desirable that the platform need not know the extension. To support this, platforms should pass the client argument of unknown extension as the authenticator argument unchanged, under the same extension identifier. The authenticator argument should be the CBOR encoding of the client argument, as specified in Section 4.2 of [RFC7049]. Clients should silently drop unknown extensions whose client argument cannot be encoded as a CBOR structure.

7.2.2 Extending client processing

Extensions may define additional processing requirements on the client platform during the creation of credentials or the generation of an assertion. In order for the RP to verify the processing took place, or if the processing has a result value that the RP needs to be aware of, the extension should specify a client data value to be included in the `clientData` structure.

The value may be any value that can be encoded as a JSON value. If any extension processed by a client defines such a value, the client should include a dictionary in `clientData` with the key `extensions`. For each such extension, the clientshould add an entry to this dictionary with the extension identifier as the key, and the extension's client data value.

7.2.3 Extending authenticator processing with signature extensions

Signature extensions that define additional authenticator processing should similarly define an authenticator data value. The value may be any data that can be encoded as a CBOR value. An authenticator that processes a signature extension that defines such a value must include it in the `authenticatorData`.

As specified in 4. Authenticator data, the authenticator data value of each processed extension is included in the extended data part of the `authenticatorData`. This part is a CBOR map, with extension identifiers as keys, and the authenticator data value of each extension as the value.

7.2.4 Example extension

This section is non-normative.

To illustrate the requirements above, consider a hypothetical extension *Geo*. This extension, if supported, lets both clients and authenticators embed their geolocation in signatures.

The extension identifier is chosen as `com.example.fido.geo`. The client argument is the constant value `true`, since the extension does not require the RP to pass any particular information to the client, other than that it requests the use of the extension. The RP sets this value in its request for an assertion:

```
var assertionPromise =
    credentials.getAssertion("SGFuIFNvbG8gc2hvdCBmaXJzdC4",
        {}, /* Empty filter */
        { 'com.example.fido.geo': true });
```

The extension defines the additional client data to be the client's location, if known, as a GeoJSON [GeoJSON] point. The client constructs the following client data:

```
{
   ...,
   'extensions': {
       'com.example.fido.geo': {
           'type': 'Point',
           'coordinates': [65.059962, -13.993041]
       }
   }
}
```

The extension also requires the client to set the authenticator parameter to the fixed value `1`.

Finally, the extension requires the authenticator to specify its geolocation in the authenticator data, if known. The extension e.g. specifies that the location shall be encoded as a two-element array of floating point numbers, encoded with CBOR. An authenticator does this by including it in the `authenticatorData`. As an example, authenticator data may be as follows (notation taken from [RFC7049]):

```
81 (hex)                             -- Flags, ED and TUP both set.
20 05 58 1F                          -- Signature counter
A1                                   -- CBOR map of one element
  68                                 -- Key 1: CBOR text string of 8 bytes
    66 69 64 6F 2E 67 65 6F          -- "fido.geo" UTF-8 string
  82                                 -- Value 1: CBOR array of two elements
    FA 42 82 1E B3                   -- Element 1: Latitude as CBOR encoded float
    FA C1 5F E3 7F                   -- Element 2: Longitude as CBOR encoded float
```

7.3 Standard extensions

This section defines standard extensions defined by the FIDO Alliance.

7.3.1 Transaction authorization

This signature extension allows for a simple form of transaction authorization. A relying party can specify a prompt string, intended for display on a trusted device on the authenticator.

Extension identifier
: `fido.txauth.simple`

Client argument
: A single UTF-8 encoded string *prompt*

Client processing
: None, except default forwarding of client argument to authenticator argument.

Authenticator argument
: The client argument encoded as a CBOR text string (major type 3).

Authenticator processing
: The authenticator must display the prompt to the user before performing the user verification / test of user presence. The authenticator may insert line breaks if needed.

Authenticator data
: A single UTF-8 string, representing the prompt *as displayed* (including any eventual line breaks).

The generic version of this extension allows images to be used as prints as well. This is allows authenticators without a font rendering engine to be used and also supports a richer visual appearance.

Extension identifier
: `fido.txauth.generic`

Client argument
: A CBOR map with one pair of data items (CBOR tagged as 0xa1). The pair of data items consists of

    1. one UTF-8 encoded string *contentType*, containing the MIME-Type of the content, e.g. "image/png"
    2. and the *content* itself, encoded as CBOR byte array.

Client processing
: None, except default forwarding of client argument to authenticator argument.

Authenticator argument
The client argument encoded as a CBOR map.

Authenticator processing
The authenticator must display the *content* to the user before performing the user verification / test of user presence. The authenticator may add other information below the *content*. No changes are allowed to the *content* itself, i.e. inside *content* boundary box.

Authenticator data
The hash value of the *content* which was displayed. The authenticator must use the same hash algorithm as it uses for the signature itself.

7.3.2 Authenticator Selection Extension

This registration extension allows a Relying Party to guide the selection of the authenticator that will be leveraged when creating the credential. It is intended primarily for Relying Parties that wish to tightly control the experience around credential creation.

Extension identifier
`fido.authn-sel` (only used during `makeCredential`)

Client argument
A sequence of AAGUIDs:

WebIDL
```
typedef sequence < AAGUID > AuthenticatorSelectionList;
```

Each AAGUID corresponds to an authenticator attestation that is acceptable to the RP for this credential creation. The list is ordered by decreasing preference. An AAGUID is defined as a DOMString, and is the globally unique identifier of the authenticator attestation being sought.

WebIDL
```
typedef DOMString AAGUID;
```

Client processing
If the client supports the Authenticator Selection Extension, it must use the first available authenticator whose AAGUID is present in the `AuthenticatorSelectionList`. If none of the available authenticators match a provided AAGUID, the client must select an authenticator from among the available authenticators to generate the credential.

Authenticator argument
There is no authenticator argument.

Authenticator processing
None.

8. IANA Considerations

This specification registers the algorithm names "S256", "S384", "S512", and "SM3" with the IANA JSON Web Algorithms registry as defined in section "Cryptographic Algorithms for Digital Signatures and MACs" in [JWA].

These names follow the naming strategy in draft-ietf-oauth-spop-15.

Algorithm Name	"S256"
Algorithm Description	The SHA256 hash algorithm.

Algorithm Usage Location(s)	"alg", i.e. used with JWS.
JOSE Implementation Requirements	Optional+
Change Controller	FIDO Alliance, Contact Us
Specification Documents	[FIPS180-4]
Algorithm Analysis Document(s)	[SP800-107r1]

Algorithm Name	"S384"
Algorithm Description	The SHA384 hash algorithm.
Algorithm Usage Location(s)	"alg", i.e. used with JWS.
JOSE Implementation Requirements	Optional
Change Controller	FIDO Alliance, Contact Us
Specification Documents	[FIPS180-4]
Algorithm Analysis Document(s)	[SP800-107r1]

Algorithm Name	"S512"
Algorithm Description	The SHA512 hash algorithm.
Algorithm Usage Location(s)	"alg", i.e. used with JWS.
JOSE Implementation Requirements	Optional+
Change Controller	FIDO Alliance, Contact Us
Specification Documents	[FIPS180-4]
Algorithm Analysis Document(s)	[SP800-107r1]

Algorithm Name	"SM3"
Algorithm Description	The SM3 hash algorithm.
Algorithm Usage Location(s)	"alg", i.e. used with JWS.
JOSE Implementation Requirements	Optional
Change Controller	FIDO Alliance, Contact Us
Specification Documents	[OSCCA-SM3]
Algorithm Analysis Document(s)	N/A

A. References

A.1 Normative references

[DOM4]

Anne van Kesteren; Aryeh Gregor; Ms2ger; Alex Russell; Robin Berjon. *W3C DOM4*. 19 November 2015. W3C Recommendation. URL: https://www.w3.org/TR/dom/

[FIDOEAP]
Reference not found.

[FIPS180-4]
FIPS PUB 180-4: Secure Hash Standard (SHS). National Institute of Standards and Technology, March 2012, URL: http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

[HTML5]
I. Hickson; R.Berjon; S. Faulkner; T. Leithead; E. D. Navara; E. O'Connor; S. Pfeiffer. *HTML5: A vocabulary and associated APIs for HTML and XHTML*. 28 October 2014. W3C Recommendation. URL: http://www.w3.org/TR/html5/

[OSCCA-SM3]
SM3 Cryptographic Hash Algorithm. December 2010. URL: http://www.oscca.gov.cn/UpFile/20101222141857786.pdf

[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

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

[WebIDL-ED]
Cameron McCormack, *Web IDL*, W3C. Editor's Draft 13 November 2014. URL: http://heycam.github.io/webidl/

A.2 Informative references

[FIDOPlatformApiReqs]
FIDO 2.0: Requirements for Native Platforms. URL: fido-platform-api-reqs.html

[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

[GeoJSON]
The GeoJSON Format Specification. URL: http://geojson.org/geojson-spec.html

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

[JWK]
M. Jones. *JSON Web Key (JWK)*. May 2015. RFC. URL: https://tools.ietf.org/html/rfc7517

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

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

[RFC6454]
A. Barth, *The Web Origin Concept (RFC 6454)*, IETF, June 2011, URL: http://www.ietf.org/rfc/rfc6454.txt

[RFC7049]
C. Bormann; P. Hoffman. *Concise Binary Object Representation (CBOR)*. October 2013. Proposed Standard. URL: https://tools.ietf.org/html/rfc7049

[RFC7159]
T. Bray, Ed.. *The JavaScript Object Notation (JSON) Data Interchange Format* March 2014. Proposed Standard. URL: https://tools.ietf.org/html/rfc7159

[SP800-107r1]
Quynh Dang, *NIST Special Publication 800-107: Recommendation for Applications Using Approved Hash Algorithms*. National Institute of Standards and Technology, August 2012, URL: http://csrc.nist.gov/publications/nistpubs/800-107-rev1/sp800-107-rev1.pdf

FIDO 2.0: Web API for accessing FIDO 2.0 credentials

FIDO Alliance Proposed Standard 04 September 2015

This version:
https://fidoalliance.org/specs/fido-v2.0-ps-20150904/fido-web-api-v2.0-ps-20150904.html
Editor:
Vijay Bharadwaj, Microsoft
Contributors:
Hubert Le Van Gong, PayPal
Dirk Balfanz, Google
Alexei Czeskis, Google
Arnar Birgisson, Google
Jeff Hodges, PayPal

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

Abstract

This document specifies an API that enables web pages to access FIDO 2.0 compliant strong cryptographic credentials through browser script. Conceptually, credentials are stored on a FIDO 2.0 authenticator, and each credential is bound to a single Relying Party. Authenticators are responsible for ensuring that no operation is performed without the user's consent. The user agent mediates access to credentials in order to preserve user privacy.

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.

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. Use Cases

This section is non-normative.

This document specifies an API for web pages to access FIDO 2.0 credentials through JavaScript, for the purpose of strongly authenticating a user. FIDO 2.0 credentials are always bound to a single FIDO Relying Party, and the API respects this requirement. Credentials created by a Relying Party can only be accessed by web origins belonging to that Relying Party. Additionally, privacy across Relying Parties must be maintained; scripts must not be able to detect any properties, or even the existence, of credentials belonging to other Relying Parties.

FIDO 2.0 credentials are located on authenticators, which can use them to perform operations subject to user consent. Broadly, authenticators are of two types:

1. **Embedded authenticators** have their operation managed by the same computing device (e.g., smart phone, tablet, desktop PC) as the user agent is running on. For instance, such an authenticator might consist of a Trusted Platform Module (TPM) or Secure Element (SE) integrated into the computing device, along with appropriate platform software to mediate access to this device's functionality.
2. **External authenticators** operate autonomously from the device running the user agent, and accessed over a transport such as Universal Serial Bus (USB), Bluetooth Low Energy (BLE) or Near Field Communications (NFC).

Note that an external authenticator may itself contain an embedded authenticator. For example, consider a smart phone that contains a FIDO 2.0 credential. The credential may be accessed by a web browser running on the phone itself. In this case the module containing the credential is functioning as an embedded authenticator. However, the credential may also be accessed over BLE by a user agent on a nearby laptop. In this latter case, the phone is functioning as an external authenticator. These modes may even be used in a single end-to-end user scenario. One such scenario is described in the remainder of this section.

1.1 Registration (embedded authenticator mode)

- On the phone:
  - User goes to example.com in the browser, and signs in using whatever method they have been using (possibly a pre-FIDO method such as a password).
  - The phone prompts, "Do you want to register this device with example.com?"
  - User agrees.
  - The phone prompts the user for a previously configured authorization gesture (PIN, biometric, etc.); the user provides this.
  - Website shows message, "Registration complete."

1.2 Authentication (external authenticator mode)

- On the laptop:
  - User goes to example.com in browser, sees an option "Sign in with your phone."
  - User chooses this option and gets a message from the browser, "Please complete this action on your phone."
- Next, on the phone:
  - User sees a discreet prompt or notification, "Sign in to example.com."
  - User selects this prompt / notification.
  - User is shown a list of their example.com identities, e.g., "Sign in as Alice / Sign in as Bob."
  - User picks an identity, is prompted for an authorization gesture (PIN, biometric, etc.) and provides this.
- Now, on the laptop:
  - Web page shows that the selected user is signed in, and navigates to the signed-in page.

1.3 Other configurations

A variety of additional use cases and configurations are also possible, including (but not limited to):

- User goes to example.com on their laptop, is guided through a flow to create and register a credential on their phone.
- User employs a FIDO 2.0 credential as described above to authorize a single transaction, such as a payment or other financial transaction.

2. 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].

This specification defines criteria for a **conforming user agent**. A user agent must behave as described in this specification in order to be considered conformant. User agents may implement algorithms given in this specification in any way desired, so long as the end result is indistinguishable from the result that would be obtained by the specification's algorithms. A conforming FIDO Credential API user agent must also be a conforming implementation of the IDL fragments of this specification, as described in the “Web IDL” specification. [WebIDL-ED]

2.1 Dependencies

This specification relies on several other underlying specifications.

HTML5
: The concept of **origin** and the **Window** interface are defined in [HTML5].

Web IDL
: Many of the interface definitions and all of the IDL in this specification depend on [WebIDL-ED]. This updated version of the Web IDL standard adds support for **Promises**, which are now the preferred mechanism for asynchronous interaction in all new web APIs.

DOM
: **DOMException** and the DOMException values used in this specification are defined in [DOM4].

FIDO External Authenticator Protocol
: This specification references methods for the client to communicate with FIDO 2.0 authenticators. These methods are specified in [FIDOEAP].

Web Cryptography API
: The **AlgorithmIdentifier** type and the method for normalizing an algorithm are defined in [WebCrypto].

3. FIDO Authenticator model

The API specified in this document implies a specific abstract functional model for a FIDO authenticator. This section describes the FIDO authenticator model. Client platforms may implement and expose this abstract model in any way desired. However, the behavior of the client's FIDO Credential API implementation, when operating on the embedded and external authenticators supported by that platform, must be indistinguishable from the behavior specified in the FIDO Credential API section.

In this abstract model, each FIDO authenticator stores some number of FIDO credentials. Each FIDO credential has an identifier which is unique (or extremely unlikely to be duplicated) among all FIDO credentials. Each credential is also associated with a FIDO Relying Party, whose identity is represented by a Relying Party Identifier (RP ID).

A client must connect to a FIDO authenticator in order to invoke any of the operations of that authenticator. This connection defines an authenticator session. A FIDO authenticator must maintain isolation between sessions. It may do this by only allowing one session to exist at any particular time, or by providing more complicated session management.

The following operations can be invoked by the client in an authenticator session.

3.1 The **authenticatorMakeCredential** operation

This operation must be invoked in an authenticator session which has no other operations in progress. It takes the following input parameters:

- The web origin of the script on whose behalf the operation is being initiated, as determined by the user agent and the client.
- The RP ID corresponding to the above web origin, as determined by the user agent and the client.
- All input parameters accepted by the `getAssertion` method, specified below.

When this operation is invoked, the authenticator obtains user consent for creating a new credential. The prompt for obtaining this consent is shown by the authenticator if it has its own output capability, or by the user agent otherwise. Once user consent is obtained, the authenticator generates the appropriate cryptographic keys and creates a new credential. It then associates the credential with the specified RP ID such that it will be able to retrieve the RP ID later, given the credential ID.

On successful completion of this operation, the authenticator returns the type and unique identifier of this new credential to the user agent.

If the user refuses consent, the authenticator returns an appropriate error status to the client.

3.2 The **authenticatorGetAssertion** operation

This operation must be invoked in an authenticator session which has no other operations in progress. It takes the following input parameters:

- The web origin of the script on whose behalf the operation is being initiated, as determined by the user agent and the client.
- The RP ID corresponding to the above web origin, as determined by the user agent and the client.
- All input parameters accepted by the `makeCredential` method, specified below.

When this method is invoked, the authenticator allows the user to select a credential from among the credentials associated with that Relying Party and matching the specified criteria, then obtains user consent for using that credential. The prompt for obtaining this consent may be shown by the authenticator if it has its own output capability, or by the user agent otherwise. Once a credential is selected and user consent is obtained, the authenticator computes a cryptographic signature using the credential's private key and constructs an assertion as specified in [FIDOSignatureFormat]. It then returns this assertion to the user agent.

If the authenticator cannot find any credential corresponding to the specified Relying Party that matches the specified criteria, it terminates the operation and returns an error.

If the user refuses consent, the authenticator returns an appropriate error status to the client.

3.3 The **authenticatorCancel** operation

This operation takes no input parameters and returns no result.

When this operation is invoked by the client in an authenticator session, it has the effect of terminating any `authenticatorMakeCredential` or `authenticatorGetAssertion` operation currently in progress in that authenticator session. The authenticator stops prompting for, or accepting, any user input related to authorizing the canceled operation. The client ignores any further responses from the authenticator for the canceled operation.

This operation is ignored if it is invoked in an authenticator session which does not have an `authenticatorMakeCredential` or `authenticatorGetAssertion` operation currently in progress.

4. FIDO Credential API

This section normatively specifies the API for creating and using FIDO 2.0 credentials. Support for deleting credentials is deliberately omitted; this is expected to be done through platform-specific user interfaces rather than from a script. The basic idea is that the credentials belong to the user and are managed by the browser and underlying platform. Scripts can (with the user's consent) request the browser to create a new credential for future use by the script's origin. Scripts can also request the user’s permission to perform authentication operations with an existing credential held by the platform. However, all such operations are mediated by the browser and/or platform on the user's behalf. At no point does the script get access to the credentials themselves; it only gets information about the credentials in the form of objects.

User agents should only expose this API to callers in**secure contexts**, as defined in [powerful-features].

In the future, this API may be integrated into a more general Web API framework for credential management, which is being worked on in the W3C. Such integration will, most likely, create intermediate interface and dictionary types, from which the types in this document will then inherit. However the experience of the FIDO developer and end user will not be substantially changed by this. In the meantime, this specification is maintained in a more minimal form for ease of review.

The API is defined by the following Web IDL fragment.

WebIDL

```
partial interface Window {
    readonly attribute FIDOCredentials fido;
};

interface FIDOCredentials {
    Promise < FIDOCredentialInfo > makeCredential (
        Account                              account,
        sequence < FIDOCredentialParameters > cryptoParameters,
        DOMString                            attestationChallenge,
        optional unsigned long               timeoutSeconds,
        optional sequence < Credential >     blacklist,
        optional FIDOExtensions              extensions
    );

    Promise < FIDOAssertion > getAssertion (
        DOMString                        assertionChallenge,
        optional unsigned long           timeoutSeconds,
        optional sequence < Credential > whitelist,
        optional FIDOExtensions          extensions
    );
};

interface FIDOCredentialInfo {
    readonly attribute Credential           credential;
    readonly attribute AlgorithmIdentifier  algorithm;
    readonly attribute any                  publicKey;
    readonly attribute AttestationStatement attestation;
};

dictionary Account {
    required DOMString rpDisplayName;
    required DOMString displayName;
    DOMString          name;
    DOMString          id;
    DOMString          imageUri;
};

dictionary FIDOCredentialParameters {
    required CredentialType        type;
    required AlgorithmIdentifier   algorithm;
};

enum CredentialType {
    "FIDO"
};

interface Credential {
    readonly attribute CredentialType type;
    readonly attribute DOMString      id;
};
```

4.1 **FIDOCredentials** Interface

This interface consists of the following methods.

4.1.1 Create a new credential (makeCredential method)

With this method, a script can request the user agent to create a new credential of a given type and persist it to the underlying platform, which may involve data storage managed by the browser or the OS. The user agent will prompt the user to approve this operation. On success, the promise will be resolved with a FIDOCredentialInfo object describing the newly created credential.

This method takes the following parameters:

- The **account** parameter specifies information about the user account for which the credential is being created. This is meant for later use by the authenticator when it needs to prompt the user to select a credential.
- The **cryptoParameters** parameter supplies information about the desired properties of the credential to be created. The sequence is ordered from most preferred to least preferred. The platform makes a best effort to create the most preferred credential that it can.
- The **attestationChallenge** parameter contains a challenge intended to be used for generating the attestation statement of the newly created credential.
- The optional **timeoutSeconds** parameter specifies a time, in seconds, that the caller is willing to wait for the call to complete. This is treated as a hint, and may be overridden by the platform.
- The optional **blacklist** parameter is intended for use by Relying Parties that wish to limit the creation of multiple credentials for the same account on a single authenticator. The platform is requested to return an error if the new credential would be created on an authenticator that also contains one of the credentials enumerated in this parameter.
- The optional **extensions** parameter contains additional parameters requesting additional processing by the client and authenticator. For example, the caller may request that only authenticators with certain capabilities be used to create the credential, or that additional information be returned in the attestation statement. Alternatively, the caller may specify an additional message that they would like the authenticator to display to the user. Extensions are defined in [FIDOSignatureFormat].

When this method is invoked, the user agent must execute the following algorithm:

1. If `timeoutSeconds` was specified, check if its value lies within a reasonable range as defined by the platform and if not, correct it to the closest value lying within that range. Set *adjustedTimeout* to this adjusted value. If `timeoutSeconds` was not specified then set *adjustedTimeout* to a platform-specific default.
2. Let *promise* be a new Promise. Return *promise* and start a timer for *adjustedTimeout* seconds. Then asynchronously continue executing the following steps.
3. Set *callerOrigin* to the `origin` of the caller. Derive the RP ID from *callerOrigin* and set *rpId* to the RP ID (see [FIDOPlatformApiReqs]).
4. Initialize *issuedRequests* to an empty list.
5. Process each element of `cryptoParameters` using the following steps:
    a. Let *current* be the currently selected element of `cryptoParameters`.
    b. If *current.type* does not contain a `CredentialType` supported by this implementation, then stop processing *current* and move on to the next element in `cryptoParameters`.
    c. Let *normalizedAlgorithm* be the result of normalizing an algorithm using the procedure defined in [WebCrypto], with *alg* set to *current.algorithm* and *op* set to "generateKey". If an error occurs during this procedure, then stop processing *current* and move on to the next element in `cryptoParameters`.
6. If `blacklist` is undefined, set it to the empty list.
7. If `extensions` was specified, process any extensions supported by this client platform, to produce the extension data that needs to be sent to the authenticator. Call this data *clientExtensions*.
8. For each embedded or external authenticator currently available on this platform: asynchronously invoke the `authenticatorMakeCredential` operation on that authenticator with *callerOrigin*, *rpId*, `account`, *current.type*, *normalizedAlgorithm*, `blacklist`, `attestationChallenge` and *clientExtensions* as parameters. Add a corresponding entry to *issuedRequests*.
9. While *issuedRequests* is not empty, perform the following actions depending upon the *adjustedTimeout* timer and responses from the authenticators:
    a. If the *adjustedTimeout* timer expires, then for each entry in *issuedRequests* invoke the `authenticatorCancel` operation on that authenticator and remove its entry from the list.
    b. If any authenticator returns a status indicating that the user cancelled the operation, delete that authenticator's entry from *issuedRequests*. For each remaining entry in *issuedRequests* invoke the `authenticatorCancel` operation on that authenticator and remove its entry from the list.
    c. If any authenticator returns an error status, delete the corresponding entry from *issuedRequests*.
    d. If any authenticator indicates success, create a new `FIDOCredentialInfo` object named *value* and populate its fields with the values returned from the authenticator. Resolve *promise* with *value* and terminate this algorithm.
10. Resolve *promise* with a `DOMException` whose name is "`NotFoundError`", and terminate this algorithm.

During the above process, the user agent should show some UI to the user to guide them in the process of selecting and authorizing an authenticator.

4.1.2 Use an existing credential (getAssertion method)

This method is used to discover and use an existing FIDO 2.0 credential, with the user's consent. The script optionally specifies some criteria to indicate what credentials are acceptable to it. The user agent and/or platform locates credentials matching the specified criteria, and guides the user to pick one that the script should be allowed to use. The user may choose not to provide a credential even if one is present, for example to maintain privacy.

This method takes the following parameters:

- The **assertionChallenge** parameter contains a string that the selected authenticator is expected to sign to produce the assertion.
- The optional **timeoutSeconds** parameter specifies a time, in seconds, that the caller is willing to wait for the call to complete. This is treated as a hint, and may be overridden by the platform.
- The optional **whitelist** member contains a list of credentials acceptable to the caller, in order of the caller's preference.
- The optional **extensions** parameter contains additional parameters requesting additional processing by the client and authenticator. For example, if transaction confirmation is sought from the user, then the prompt string would be included in an extension. Extensions are defined in a companion specification.

When this method is invoked, the user agent must execute the following algorithm:

1. If `timeoutSeconds` was specified, check if its value lies within a reasonable range as defined by the platform and if not, correct it to the closest value lying within that range. Set *adjustedTimeout* to this adjusted value. If `timeoutSeconds` was not specified then set *adjustedTimeout* to a platform-specific default.
2. Let *promise* be a new Promise. Return *promise* and start a timer for *adjustedTimeout* seconds. Then asynchronously continue executing the following steps.
3. Set *callerOrigin* to the `origin` of the caller. Derive the RP ID from *callerOrigin* and set *rpId* to the RP ID (see [FIDOPlatformApiReqs]).
4. Initialize *issuedRequests* to an empty list.
5. If `extensions` was specified, process any extensions supported by this client platform, to produce the extension data that needs to be sent to the authenticator. Call this data *clientExtensions*.
6. For each embedded or external authenticator currently available on this platform, perform the following steps:
    a. If `whitelist` is undefined or empty, let *credentialList* be a list containing a single wildcard entry.
    b. If `whitelist` is defined and non-empty, optionally execute a platform-specific procedure to determine which of these credentials can possibly be present on this authenticator. Set *credentialList* to this filtered list. If *credentialList* is empty, ignore this authenticator and do not perform any of the following per-authenticator steps.
    c. Asynchronously invoke the `authenticatorGetAssertion` operation on this authenticator with *callerOrigin*, *rpId*, `assertionChallenge`, *credentialList*, and *clientExtensions* as parameters.
    d. Add an entry to *issuedRequests*, corresponding to this request.
7. While *issuedRequests* is not empty, perform the following actions depending upon the *adjustedTimeout* timer and responses from the authenticators:
    a. If the timer for *adjustedTimeout* expires, then for each entry in *issuedRequests* invoke the `authenticatorCancel` operation on that authenticator and remove its entry from the list.
    b. If any authenticator returns a status indicating that the user cancelled the operation, delete that authenticator's entry from *issuedRequests*. For each remaining entry in *issuedRequests* invoke the `authenticatorCancel` operation on that authenticator, and remove its entry from the list.
    c. If any authenticator returns an error status, delete the corresponding entry from *issuedRequests*.
    d. If any authenticator returns success, create a new `FIDOAssertion` object named *value* and populate its fields with the values returned from the authenticator. Resolve *promise* with `value` and terminate this algorithm.
8. Resolve *promise* with a `DOMException` whose name is "`NotFoundError`", and terminate this algorithm.

During the above process, the user agent should show some UI to the user to guide them in the process of selecting and authorizing an authenticator with which to complete the operation.

4.2 **FIDOCredentialInfo** Interface

This interface represents a newly-created FIDO credential. It contains information about the credential that can be used to locate it later for use, and also contains metadata that can be used by the FIDO Relying Party to assess the strength of the credential during registration.

The **credential** attribute contains a unique identifier for the credential represented by this object.

The **algorithm** attribute contains the cryptographic algorithm associated with the credential, in the format defined in [WebCrypto].

The **publicKey** attribute contains the public key associated with the credential, represented as a JsonWebKey structure as defined in [WebCrypto].

The **attestation** attribute contains a key attestation statement returned by the authenticator. This provides information about the credential and the authenticator it is held in, such as the level of security assurance provided by the authenticator.

4.3 User Account Information (dictionary **Account**)

This dictionary is used by the caller to specify information about the user account and Relying Party with which a credential is to be associated. It is intended to help the authenticator in providing a friendly credential selection interface for the user.

The **rpDisplayName** member contains the friendly name of the Relying Party, such as "Google", "Microsoft" or "PayPal".

The **displayName** member contains the friendly name associated with the user account by the Relying Party, such as "John P. Smith".

The **name** member contains a detailed name for the account, such as "john.p.smith@example.com".

The **id** member contains an identifier for the account, stored for the use of the Relying Party. This is not meant to be displayed to the user.

The **imageUri** member contains a URI that resolves to the user's account image. This may be a URL that can be used to retrieve the user's current avatar, or a data URI that contains the image data.

4.4 Parameters for Credential Generation (dictionary **FIDOCredentialParameters**)

This dictionary is used to supply additional parameters when creating a new credential.

The **type** member specifies the type of credential to be created.

The **algorithm** member specifies the cryptographic algorithm with which the newly generated credential will be used.

4.5 Supporting Data Structures

The FIDO credential type uses certain data structures that are specified in supporting documents. These are as follows.

4.5.1 Credential Type enumeration (enum CredentialType)

This enumeration defines the valid credential types. It is an extension point; values may be added to it in the future, as more credential types are defined. The values of this enumeration are used for versioning the FIDO assertion and attestation statement according to the type of the authenticator.

Currently one credential type is defined, namely "**FIDO**", the FIDO 2.0 credential type.

4.5.2 Unique Identifier for Credential (interface Credential)

This interface contains the attributes that are returned to the caller when a new credential is created, and can be used later by the caller to select a credential for use.

The **type** attribute indicates the specification and version that this credential conforms to.

The **id** attribute contains an identifier for the credential, chosen by the platform with help from the authenticator. This identifier is used to look up credentials for use, and is therefore expected to be globally unique with high probability across all credentials of the same type. This API does not constrain the format or length of this identifier, except that it must be sufficient for the platform to uniquely select a key. For example, an authenticator without on-board storage may create identifiers that consist of the key material wrapped with a key that is burned into the authenticator.

4.5.3 Cryptographic Algorithm Identifier (type AlgorithmIdentifier)

A string or dictionary identifying a cryptographic algorithm and optionally a set of parameters for that algorithm. This type is defined in [WebCrypto].

4.5.4 FIDO Assertion (interface FIDOAssertion)

FIDO 2.0 credentials produce a cryptographic signature that provides proof of possession of a private key as well as evidence of user consent to a specific transaction. The structure of these signatures is defined in [FIDOSignatureFormat].

4.5.5 FIDO Assertion Extensions (dictionary FIDOExtensions)

This is a dictionary containing zero or more extensions as defined in [FIDOSignatureFormat]. An extension is an additional parameter that can be passed to the getAssertion method and triggers some additional processing by the client platform and/or the authenticator.

If the caller wants to pass extensions to the platform, it should do so by adding one entry per extension to this `FIDOExtensions` dictionary with the extension identifier as the key, and the extension's value as the value (see [FIDOSignatureFormat] for details).

4.5.6 Key Attestation Statement (interface AttestationStatement)

FIDO 2.0 authenticators also provide some form of key attestation. The basic requirement is that the authenticator can produce, for each credential public key, attestation information that can be verified by a Relying Party. Typically this information contains a signature by an attesting key over the attested public key and a challenge, as well as a certificate or similar information providing provenance information for the attesting key, enabling a trust decision to be made. The structure of these attestation statements is defined in [FIDOKeyAttestation].

5. Sample scenarios

This section is non-normative.

In this section, we walk through some events in the lifecycle of a FIDO 2.0 credential, along with the corresponding sample code for using this API. Note that this is an example flow, and does not limit the scope of how the API can be used.

As was the case in earlier sections, this flow focuses on a use case involving an external first-factor authenticator with its own display. One example of such an authenticator would be a smart phone. Other authenticator types are also supported by this API, subject to implementation by the platform. For instance, this flow also works without modification for the case of an authenticator that is embedded in the client platform. The flow also works for the case of an external authenticator without its own display (similar to a smart card) subject to specific implementation considerations. Specifically, the client platform needs to display any prompts that would otherwise be shown by the authenticator, and the authenticator needs to allow the client platform to enumerate all the authenticator's credentials so that the client can have information to show appropriate prompts.

5.1 Registration

This is the first time flow, when a new credential is created and registered with the server.

1. The user visits example.com, which serves up a script. At this point, the user must already be logged in using a legacy username and password, or additional authenticator, or other means acceptable to the Relying Party.
2. The Relying Party script runs the code snippet below.
3. The client platform searches for and locates the external authenticator.
4. The client platform connects to the external authenticator, performing any pairing actions if necessary.
5. The external authenticator shows appropriate UI for the user to select the authenticator on which the new credential will be created, and obtains a biometric or other authorization gesture from the user.
6. The external authenticator returns a response to the client platform, which in turn returns a response to the RP script. If the user declined to select an authenticator or provide authorization, an appropriate error is returned.
7. If a new credential was created,
    a. The RP script sends the newly generated public key to the server, along with additional information about public key such as attestation that it is held in trusted hardware.
    b. The server stores the public key in its database and associates it with the user as well as with the strength of authentication indicated by attestation, also storing a friendly name for later use.
    c. The script may store data such as the credential ID in local storage, to improve future UX by narrowing the choice of credential for the user.

The sample code for generating and registering a new key follows:

EXAMPLE 1

```
var fidoAPI = window.fido;

if (!fidoAPI) { /* Platform not capable. Handle error. */ }

var userAccountInformation = {
  rpDisplayName: "PayPal",
  displayName: "John P. Smith",
  name: "johnpsmith@gmail.com",
  id: "1098237235409872";
  imageUri: "https://pics.paypal.com/00/p/aBjjjpqPb.png"
};

// This RP will accept either an ES256 or RS256 credential, but
// prefers an ES256 credential.
var cryptoParams = [
  {
    type: "FIDO",
    algorithm: "ES256",
  },
  {
    type: "FIDO",
    algorithm: "RS256",
  }
];
var challenge = "Y2xpbWIgYSBtb3VudGFpbg";
var timeoutSeconds = 300;  // 5 minutes
var blacklist = [];  // No blacklist
var extensions = {"fido.location": true};  // Include location information in attestation


// Note: The following call will cause the authenticator to display UI.
fidoAPI.makeCredential(userAccountInformation, cryptoParams, challenge,
                       timeoutSeconds, blacklist, extensions)
  .then(function (newCredentialInfo) {
    // Send new credential info to server for verification and registration.
}).catch(function (err) {
    // No acceptable authenticator or user refused consent. Handle appropriately.
});
```

5.2 Authentication

This is the flow when a user with an already registered credential visits a website and wants to authenticate using the credential.

1. The user visits example.com, which serves up a script.
2. The script asks the client platform for a FIDO identity assertion, providing as much information as possible to narrow the choice of acceptable credentials for the user. This may be obtained from the data that was stored locally after registration, or by other means such as prompting the user for a username.
3. The Relying Party script runs one of the code snippets below.
4. The client platform searches for and locates the external authenticator.
5. The client platform connects to the external authenticator, performing any pairing actions if necessary.
6. The external authenticator presents the user with a notification that their attention is required. On opening the notification, the user is shown a friendly selection menu of acceptable credentials using the account information provided when creating the credentials, along with some information on the origin that is requesting these keys.
7. The authenticator obtains a biometric or other authorization gesture from the user.
8. The external authenticator returns a response to the client platform, which in turn returns a response to the RP script. If the user declined to select a credential or provide an authorization, an appropriate error is returned.
9. If an assertion was successfully generated and returned,
    a. The script sends the assertion to the server.
    b. The server examines the assertion and validates that it was correctly generated. If so, it looks up the identity associated with the associated public key; that identity is now authenticated. If the public key is not recognized by the server (e.g., deregistered by server due to inactivity) then the authentication has failed; each Relying Party will handle this in its own way.
    c. The server now does whatever it would otherwise do upon successful authentication — return a success page, set authentication cookies, etc.

If the Relying Party script does not have any hints available (e.g., from locally stored data) to help it narrow the list of credentials, then the sample code for performing such an authentication might look like this:

EXAMPLE 2

```
  var fidoAPI = window.fido;

  if (!fidoAPI) { /* Platform not capable. Handle error. */ }

  var challenge = "Y2xpbWIgYSBtb3VudGFpbg";
  var timeoutSeconds = 300;  // 5 minutes
  var whitelist = [{ type: "FIDO" }];

  fidoAPI.getAssertion(challenge, timeoutSeconds, whitelist)
    .then(function (assertion) {
      // Send assertion to server for verification
  }).catch(function (err) {
      // No acceptable credential or user refused consent. Handle appropriately.
  });
```

On the other hand, if the Relying Party script has some hints to help it narrow the list of credentials, then the sample code for performing such an authentication might look like the following. Note that this sample also demonstrates how to use the extension for transaction authorization.

EXAMPLE 3

```
var fidoAPI = window.fido;

if (!fidoAPI) { /* Platform not capable. Handle error. */ }

var challenge = "Y2xpbWIgYSBtb3VudGFpbg";
var timeoutSeconds = 300;  // 5 minutes
var acceptableCredential1 = {
  type: "FIDO",
  id: "ISEhISEhIWhpIHRoZXJlISEhISEhIQo=",
};
var acceptableCredential2 = {
  type: "FIDO",
  id: "cm9zZXMgYXJlIHJlZCwgdmlvbGV0cyBhcmUgYmx1ZQo=",
};
var whitelist = [acceptableCredential1, acceptableCredential2];
var extensions = { 'fido.txauth.simple': "Wave your hands in the air like you just don't care" };

fidoAPI.getAssertion(challenge, timeoutSeconds, whitelist, extensions)
  .then(function (assertion) {
    // Send assertion to server for verification
}).catch(function (err) {
    // No acceptable credential or user refused consent. Handle appropriately.
});
```

5.3 Decommissioning

The following are possible situations in which decommissioning a credential might be desired. Note that all of these are handled on the server side and do not need support from the API specified here.

- Possibility #1 — user reports the credential as lost.
  - User goes to server.example.net, authenticates and follows a link to report a lost/stolen device.
  - Server returns a page showing the list of registered credentials with friendly names as configured during registration.
  - User selects a credential and the server deletes it from its database.
  - In future, Relying Party script does not specify this credential in any list of acceptable credentials, and assertions signed by this credential are rejected.
- Possibility #2 — server deregisters the credential due to inactivity.
  - Server deletes credential from its database during maintenance activity.
  - In the future, the Relying Party script does not specify this credential in any list of acceptable credentials, and assertions signed by this credential are rejected.
- Possibility #3 — user deletes the credential from the device.
  - User employs a device-specific method (e.g., device settings UI) to delete a credential from their device.
  - From this point on, this credential will not appear in any selection prompts, and no assertions can be generated with it.
  - Sometime later, the server deregisters this credential due to inactivity.

6. Acknowledgements

This section is non-normative.

We would like to thank the following for their contributions to, and thorough review of, this specification: Jing Jin, Michael B. Jones, Rolf Lindemann.

A. References

A.1 Normative references

[DOM4]
Anne van Kesteren; Aryeh Gregor; Ms2ger; Alex Russell; Robin Berjon. *W3C DOM4*. 19 November 2015. W3C Recommendation. URL: https://www.w3.org/TR/dom/

[FIDOEAP]
Reference not found.

[FIDOKeyAttestation]
FIDO 2.0: Key attestation format. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20150904/fido-key-attestation.html

[FIDOPlatformApiReqs]
FIDO 2.0: Requirements for Native Platforms. URL: fido-platform-api-reqs.html

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

[HTML5]
I. Hickson; R.Berjon; S. Faulkner; T. Leithead; E. D. Navara; E. O'Connor; S. Pfeiffer. *HTML5: A vocabulary and associated APIs for HTML and XHTML*. 28 October 2014. W3C Recommendation. URL: http://www.w3.org/TR/html5/

[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

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

[WebIDL-ED]
Cameron McCormack, *Web IDL*, W3C. Editor's Draft 13 November 2014. URL: http://heycam.github.io/webidl/

A.2 Informative references

[powerful-features]
Mike West. *Secure Contexts*. 15 September 2016. W3C Candidate Recommendation. URL: https://www.w3.org/TR/secure-contexts/