Annotation Summary

59 annotations on 24 pages by Adam Powers

x26 x33

Abstract

This jumps right into using jargon -- given the potential audience, do we want to ease them into he jargon?

This specification defines an API that enables web pages to access WebAuthn compliant strong

#1 p.1

Safe to assume that readers are familiar with scoped credentials?

This specification defines an API for web pages to access scoped credentials through JavaScript, for the purpose of strongly authenticating a user. Scoped credentials are always scoped to a single

#2 p.5

---

§ 1.1. Registration (embedded authenticator mode)

Diagrams?

- On the phone:

#3 p.5

---

request the browser to create a new credential for future use by the WebAuthn Relying Party. Scripts can also request the user's permission to perform authentication operations with an existing credential. All such operations are performed in the authenticator and 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.

?

The security properties of this API are provided by the client and the authenticator working

#4 p.8

```
partial interface Window {
    readonly attribute WebAuthentication webauthn;
};
```

Is there any rhyme or reason to the order of this WebIDL?

#5 p.9

```
interface ScopedCredentialInfo {
    readonly attribute Credential             credential;
    readonly attribute any                    publicKey;
    readonly attribute WebAuthnAttestation    attestation;
};

dictionary Account {
    required DOMString rpDisplayName;
    required DOMString displayName;
    DOMString          name;
```

If attestation is null, there would be no clientDataHash. Negative security ramifications?

#6 p.9

```
};

interface WebAuthnAttestation {
    readonly    attribute DOMString      type;
    readonly    attribute ArrayBuffer    clientData;
    readonly    attribute any            statement;
};
```

enum?

#7 p.10

---

§ **3.1.1. 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 `ScopedCredentialInfo` object describing the newly created credential.

This method takes the following parameters:

#8 p.10

---

4. Initialize *issuedRequests* to an empty list.
5. Process each element of cryptoParameters using the following steps, to produce a new sequence `normalizedParameters`:
   - Let *current* be the currently selected element of cryptoParameters.

Additional description of what normalization is trying to accomplish would be helpful

#9 p.11

- 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.
- 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.
- If any authenticator returns an error status, delete the corresponding entry from *issuedRequests*.

The intent is that cancel on one results in cancelling all?

#10 p.12

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.

...but requests are sent to all authenticators in #8 above?

§ **3.1.2. Use an existing credential (*getAssertion()* method)**

#11 p.13

---

6. For each embedded or external authenticator currently available on this platform, perform the following steps:
   - If whitelist is undefined or empty, let *credentialList* be a list containing a single wildcard entry.
   - 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.
   - Asynchronously invoke the authenticatorGetAssertion operation on this authenticator with *callerOrigin*, *rpId*, assertionChallenge, *credentialList*, and *clientExtensions* as parameters.

First mention of wildcards? Where are they defined?

Implies cred id was stored in makeCred? Maybe some hint in makeCred?

#12 p.14

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.

Described in section ###?

#13 p.15

URL that can be used to retrieve an image containing the user's current avatar, or a data URI that contains the image data.

Other fields okay?

§ 3.4. Parameters for Credential Generation (dictionary

#14 p.15

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

will = should?

§ 3.5. WebAuthn Assertion (interface ***WebAuthnAssertion***)

#15 p.16

---

may?

Authenticators also provide some form of attestation. The basic requirement is that the authenticator can produce, for each credential public key, attestation information that can be verified by a WebAuthn 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

#16 p.17

---

§ **4.1.1. The *authenticatorMakeCredential* operation**

Should clientDataHash be passed in, or is this assuming that the authn calculates it?

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

#17 p.19

---

with the same type across all authenticators. 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.

Doesn't mention creating an attestation or processing extensions?

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

#18 p.19

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

Really vague. I assume that's okay?

§ **4.1.2. The *authenticatorGetAssertion* operation**

#19 p.20

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

Any considerations for rolling back state? (counters, keys, etc.)

§ 4.2. Signature Format

#20 p.21

- 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.

Refs?

#21 p.21

```
    WebAuthnExtensions            extensions;
};
```

ArrayBuffer?

The ***challenge*** member contains the base64url encoding of the challenge provided by the RP.

The ***facet*** member contains the fully qualified web origin of the requester, as provided to the authenticator by the client, in the syntax defined by [RFC6454].

#22 p.22

---

	extensions.
1-4	Signature counter (`signCount`), 32
5-	Extension-defined authenticator data. This is a CBOR [RFC7049] map with extension identifiers as keys, and extension authenticator data values as values. See §5 WebAuthn Extensions for details.

first mention of CBOR. Remove, add some clarifying text, or a pointer to the extensions section?

#23 p.23

---

bytes long. If the ED flag is set, it is 5 bytes plus the CBOR map that follows.

§ **4.2.3. Generating a signature**

What is the relationship between this and authenticatorMake

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

#24 p.24

---

data. Thus, an authenticator SHALL compute a signature over the concatenation of the *authenticatorData* and the *clientDataHash*.

There's no mention of incrementing the counter? Also, should authenticatorMakeCredential mention storing / retrieving it?

#25 p.24

The authenticator MUST return both the authenticatorData and the raw signature back to the client. The client, in turn, MUST return clientDataJSON, authenticatorData and the signature to the RP. The clientDataJSON is returned in the `clientData` member of the `WebAuthnAssertion` and `AttestationStatement` structures.

WebAuthnAttestation?

§ 4.3. Credential Attestation Statements

Thinking ahead to future attestation formats, is there a requirement that they MUST sign over clientDataHash?

An attestation statement is a specific type of signature, which contains statements about a credential

#26 p.25

---

cryptographically valid.

Attestation types are orthogonal to attestation models, i.e. attestation types in general are not restricted to a single attestation model. Broadly speaking, attestation types pertain to the syntax of the attestation statement, while attestation models pertain to the semantics.

Maybe a concrete example?

§ **4.3.1. Attestation Models**

#27 p.26

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

Refs or kill?

Surrogate Basic Attestation

In the case of surrogate basic attestation [UAFProtocol], the Authenticator doesn't have any

#28 p.26

If attestation certificate validation fails due to a revoked intermediate attestation CA certificate, and the WebAuthn Relying Party's policy requires rejecting the registration/authentication request in these situations, then it is recommended that the WebAuthn Relying Party also un-registers (or marks as "surrogate attestation" (see §4.3.1 Attestation Models), policy permitting) scoped 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 WebAuthn Relying Parties remember intermediate attestation CA certificates during Authenticator registration in order to un-

?

#29 p.37

---

extensions.

- Authenticator processing, and the authenticatorData structure, for signature extensions.

Or the authn may spontaneously create extensions

When requesting an assertion for a scoped credential, a WebAuthn Relying Party 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

#30 p.38

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.

caps?

#31 p.40

---

§ 8. Sample scenarios

Why aren't these in the use cases section?

This section is not normative.

#32 p.49

Web Authentication: A Web API for accessing scoped credentials

Editor's Draft, 13 May 2016

This version:

http://w3c.github.io/webauthn/

Latest published version:

http://www.w3.org/TR/webauthn/

Editors:

Vijay Bharadwaj (Microsoft)
Hubert Le Van Gong (PayPal)
Dirk Balfanz (Google)
Alexei Czeskis (Google)
Arnar Birgisson (Google)
Jeff Hodges (PayPal)
Michael B. Jones (Microsoft)
Rolf Lindemann (Nok Nok Labs)

Abstract

This jumps right into using jargon -- given the potential audience, do we want to ease them into he jargon?

This specification defines an API that enables web pages to access WebAuthn compliant strong cryptographic credentials through browser script. Conceptually, one or more credentials are stored on an authenticator, and each credential is scoped 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. Authenticators use attestation to provide cryptographic proof of their properties to the relying party. This specification also describes a functional model of a WebAuthn compliant authenticator, including its signature and attestation functionality.

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 W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

This document was published by the Web Authentication Working Group as an Editors' Draft. This document is intended to become a W3C Recommendation. Feedback and comments on this specification are welcome. Please use Github issues. Discussions may also be found in the public-webauthn@w3.org archives.

Publication as an Editors' Draft does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

This document is governed by the 1 September 2015 W3C Process Document.

Table of Contents

§ 1. Use Cases

This section is not normative.

Safe to assume that readers are familiar with scoped credentials?

This specification defines an API for web pages to access scoped credentials through JavaScript, for the purpose of strongly authenticating a user. Scoped credentials are always scoped to a single WebAuthn Relying Party. This scoping is enforced jointly by the User Agent implementing the Web Authentication API and the authenticator that holds the credential, by constraining the availability and usage of credentials. Scoped credentials created by a WebAuthn Relying Party can only be accessed by web origins belonging to that WebAuthn Relying Party. Additionally, privacy across WebAuthn Relying Parties must be maintained; scripts must not be able to detect any properties, or even the existence, of scoped credentials belonging to other WebAuthn Relying Parties.

Scoped 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 scoped 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)

Diagrams?

- On the phone:
  - User goes to example.com in the browser and signs in to an existing account using whatever method they have been using (possibly a legacy method such as a password), or creates a new account.
  - 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 scoped credential as described above to authorize a single transaction, such as a payment or other financial transaction.

§ 2. Conformance

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 Web Authentication API User Agent MUST also be a conforming implementation of the IDL fragments of this specification, as described in the "Web IDL" specification. [WebIDL-1]

This specification also defines a model of a Web Authentication compliant authenticator. This is a set of functional and security requirements for an authenticator to be usable by a User Agent that implements the Web Authentication API. As described in §1 Use Cases, the authenticator itself may be implemented in the operating system underlying the User Agent, or in external hardware, or a combination of both.

§ 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-1]. 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].

Web Cryptography API
: The ***AlgorithmIdentifier*** type and the method for normalizing an algorithm are defined in Web Cryptography API §algorithm-dictionary.

  The ***JsonWebKey*** dictionary for representing cryptographic keys is defined in Web Cryptography API §JsonWebKey-dictionary.

Base64url encoding
: 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) [RFC7515].

§ 3. Web Authentication API

This section normatively specifies the API for creating and using scoped 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 an authenticator, with which the WebAuthn Relying Party interacts through the client (consisting of the browser and underlying OS platform). Scripts can (with the user's consent) request the browser to create a new credential for future use by the WebAuthn Relying Party. Scripts can also request the user's permission to perform authentication operations with an existing credential. All such operations are performed in the authenticator and 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.

The security properties of this API are provided by the client and the authenticator working together. The authenticator, which holds and manages credentials, ensures that all operations are scoped to a particular web origin, and cannot be replayed against a different origin, by incorporating the origin in its responses. Specifically, as defined in §4.2 Signature Format, the full origin of the requester is included, and signed over, in the attestation statement produced when a new credential is created as well as in all assertions produced by WebAuthn credentials.

Additionally, to maintain user privacy and prevent malicious WebAuthn Relying Parties from probing for the presence of credentials belonging to other WebAuthn Relying Parties, each credential is also associated with a Relying Party Identifier, or RP ID. This RP ID is provided by the client to the authenticator for all operations, and the authenticator ensures that credentials created by a WebAuthn Relying Party can only be used in operations requested by the same RP ID. Separating the origin from the RP ID in this way allows the API to be used in cases where a single WebAuthn Relying Party maintains multiple web origins.

The client facilitates these security measures by providing correct web origins and RP IDs to the authenticator for each operation. Since this is an integral part of the WebAuthn security model, user agents SHOULD only expose this API to callers in ***secure contexts***, as defined in [secure-contexts].

The API is defined by the following Web IDL fragment.

```
partial interface Window {
    readonly attribute WebAuthentication webauthn;
};

interface WebAuthentication {
    Promise < ScopedCredentialInfo > makeCredential (
        Account                                 accountInformation,
        sequence < ScopedCredentialParameters > cryptoParameters,
        BufferSource                            attestationChallenge,
        optional unsigned long                  credentialTimeoutSeconds
        optional sequence < Credential >        blacklist,
        optional WebAuthnExtensions             credentialExtensions
    );

    Promise < WebAuthnAssertion > getAssertion (
        BufferSource                       assertionChallenge,
        optional unsigned long             assertionTimeoutSeconds,
        optional sequence < Credential > whitelist,
        optional WebAuthnExtensions        assertionExtensions
    );
};

interface ScopedCredentialInfo {
    readonly attribute Credential           credential;
    readonly attribute any                  publicKey;
    readonly attribute WebAuthnAttestation  attestation;
};

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

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

interface WebAuthnAssertion {
    readonly attribute Credential  credential;
```

Is there any rhyme or reason to the order of this WebIDL?

If attestation is null, there would be no clientDataHash. Negative security ramifications?

```
    readonly attribute ArrayBuffer clientData;
    readonly attribute ArrayBuffer authenticatorData;
    readonly attribute ArrayBuffer signature;
};

dictionary WebAuthnExtensions {
};

interface WebAuthnAttestation {
    readonly    attribute DOMString     type;
    readonly    attribute ArrayBuffer   clientData;
    readonly    attribute any           statement;
};

enum CredentialType {
    "ScopedCred"
};

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

enum?

§ 3.1. *WebAuthentication* Interface

This interface has two methods, which are described in the following subsections.

§ 3.1.1. 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 `ScopedCredentialInfo` object describing the newly created credential.

This method takes the following parameters:

- The ***accountInformation*** 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 ***credentialTimeoutSeconds*** 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 WebAuthn 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 ***credentialExtensions*** 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 §5 WebAuthn Extensions.

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

1. If credentialTimeoutSeconds 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 credentialTimeoutSeconds 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* by computing the "public suffix + 1" or "PS+1" (which is also referred to as the "Effective Top-Level Domain plus One" or "eTLD+1") part of *callerOrigin* [PSL]. Set *rpId* to the RP ID.
4. Initialize *issuedRequests* to an empty list.
5. Process each element of cryptoParameters using the following steps, to produce a new sequence `normalizedParameters`: Additional description of what normalization is trying to accomplish would be helpful
    - Let *current* be the currently selected element of cryptoParameters.

- 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.
- Let `normalizedAlgorithm` be the result of normalizing an algorithm using the procedure defined in [WebCryptoAPI], 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.
- Add a new object of type `ScopedCredentialParameters` to `normalizedParameters`, with *type* set to `current.type` and *algorithm* set to `normalizedAlgorithm`.

6. If blacklist is undefined, set it to the empty list.
7. If credentialExtensions 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*, accountInformation, `normalizedParameters`, 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:
   - 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.
   - 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.

     The intent is that cancel on one results in cancelling all?
   - If any authenticator returns an error status, delete the corresponding entry from *issuedRequests*.
   - If any authenticator indicates success, create a new `ScopedCredentialInfo` 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.

...but requests are sent to all authenticators in #8 above?

§ 3.1.2. Use an existing credential (*getAssertion()* method)

This method is used to discover and use an existing scoped 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 challenge that the selected authenticator is expected to sign to produce the assertion.
- The optional ***assertionTimeoutSeconds*** 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 ***assertionExtensions*** 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 assertionTimeoutSeconds 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 assertionTimeoutSeconds 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* by computing the "public suffix + 1" or "PS+1" (which is also referred to as the "Effective Top-Level Domain plus One" or "eTLD+1") part of *callerOrigin* [PSL]. Set *rpId* to the RP ID.
4. Initialize *issuedRequests* to an empty list.

5. If assertionExtensions 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:

   - If whitelist is undefined or empty, let *credentialList* be a list containing a single wildcard entry.

   - 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.

   - Asynchronously invoke the authenticatorGetAssertion operation on this authenticator with *callerOrigin*, *rpId*, assertionChallenge, *credentialList*, and *clientExtensions* as parameters.

   - 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:

   - 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.

   - 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.

   - If any authenticator returns an error status, delete the corresponding entry from *issuedRequests*.

   - If any authenticator returns success, create a new `WebAuthnAssertion` 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.

§ 3.2. *ScopedCredentialInfo* Interface

This interface represents a newly-created scoped 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 WebAuthn 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 ***publicKey*** attribute contains the public key associated with the credential, represented as a JsonWebKey structure as defined in Web Cryptography API §JsonWebKey-dictionary.

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. **Described in section ###?**

§ 3.3. User Account Information (dictionary *Account*)

This dictionary is used by the caller to specify information about the user account and WebAuthn 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 WebAuthn Relying Party, such as "Acme Corporation", "Widgets Inc" or "Awesome Site".

The ***displayName*** member contains the friendly name associated with the user account by the WebAuthn 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 WebAuthn Relying Party. This is not meant to be displayed to the user.

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

Other fields okay?

§ 3.4. Parameters for Credential Generation (dictionary *ScopedCredentialParameters*)

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.

will = should?

§ 3.5. WebAuthn Assertion (interface ***WebAuthnAssertion***)

Scoped 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 as follows.

The ***credential*** member represents the credential that was used to generate this assertion.

The ***clientData*** member contains the parameters sent to the authenticator by the client, in serialized form. See §4.2.1 Client data used in WebAuthn signatures (dictionary ClientData) for the format of this parameter and how it is generated.

The ***authenticatorData*** member contains the serialized data returned by the authenticator. See §4.2.2 Authenticator data.

The ***signature*** member contains the raw signature returned from the authenticator. See §4.2.3 Generating a signature.

§ 3.6. WebAuthn Assertion Extensions (dictionary ***WebAuthnExtensions***)

This is a dictionary containing zero or more extensions as defined in §5 WebAuthn Extensions. 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 dictionary with the extension identifier as the key, and the extension's value as the value (see §4.2 Signature Format for details).

§ 3.7. Credential Attestation Statement (interface ***WebAuthnAttestation***)

may?

Authenticators also provide some form of attestation. The basic requirement is that the authenticator can produce, for each credential public key, attestation information that can be verified by a WebAuthn 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 ***type*** member specifies the type of attestation statement contained in this structure. This specification defines a number of attestation types, in §4.3.2 Defined Attestation Types. Other attestation types may be defined in later versions of this specification.

The ***clientData*** member contains the clientDataJSON (see §4.2 Signature Format). The exact JSON encoding must be preserved as the hash (clientDataHash) has been computed over it.

The ***statement*** element contains the actual attestation statement. The structure of this object depends on the attestation type. For more details, see §4.3 Credential Attestation Statements.

This attestation statement is delivered to the WebAuthn Relying Party by the WebAuthn Relying Party's script running on the client, using methods outside the scope of this specification. It contains all the information that the WebAuthn Relying Party's server requires to validate the statement, as well as to decode and validate the bindings of both the client and authenticator data.

§ 3.8. Supporting Data Structures

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

§ 3.8.1. 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 WebAuthn assertion and attestation statement according to the type of the authenticator.

Currently one credential type is defined, namely "***ScopedCred***".

§ 3.8.2. 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 contains a value of type `CredentialType`, indicating 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, across all authenticators. 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.

§ 3.8.3. 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 [WebCryptoAPI].

§ 4. WebAuthn Authenticator model

The API defined in this specification implies a specific abstract functional model for an authenticator. This section describes the authenticator model. Client platforms may implement and expose this abstract model in any way desired. However, the behavior of the client's Web Authentication API implementation, when operating on the embedded and external authenticators supported by that platform, MUST be indistinguishable from the behavior specified in §3 Web Authentication API.

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

§ 4.1. Authenticator operations

A client must connect to an authenticator in order to invoke any of the operations of that authenticator. This connection defines an authenticator session. An 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.

§ 4.1.1. The *authenticatorMakeCredential* operation

Should clientDataHash be passed in, or is this assuming that the authn calculates it?

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.
- The `Account` information provided by the WebAuthn Relying Party.
- The `CredentialType` requested by the WebAuthn Relying Party.
- The cryptographic parameters requested by the WebAuthn Relying Party, with the cryptographic algorithms normalized as per the procedure in Web Cryptography API §algorithm-normalization-normalize-an-algorithm.
- A list of `Credential` objects provided by the WebAuthn Relying Party with the intention that, if any of these are known to the authenticator, it should not create a new credential.
- A challenge provided by the WebAuthn Relying Party to assure freshness of the attestation statement of the new credential.
- Extension data created by the client based on the extensions requested by the WebAuthn Relying Party.

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 also generates an identifier for the credential, such that this identifier is globally unique with high probability across all credentials with the same type across all authenticators. 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.

Doesn't mention creating an attestation or processing extensions?

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.

Really vague. I assume that's okay?

§ 4.1.2. 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.
- A challenge provided by the WebAuthn Relying Party to assure freshness of the assertion produced.
- A list of credentials acceptable to the WebAuthn Relying Party (possibly filtered by the client).
- Extension data created by the client based on the extensions requested by the WebAuthn Relying Party.

When this method is invoked, the authenticator allows the user to select a credential from among the credentials associated with that WebAuthn 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 §4.2 Signature Format. It then returns this assertion to the user agent.

If the authenticator cannot find any credential corresponding to the specified WebAuthn 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.

§ 4.1.3. 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.

Any considerations for rolling back state? (counters, keys, etc.)

§ 4.2. Signature Format

WebAuthn 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 WebAuthn can be divided into three layers:

1. The WebAuthn Relying Party (RP), which uses the WebAuthn services. The RP consists of a server component and a web-application running in a browser.
2. The WebAuthn Client platform, which consists of the User Agent and the OS and device on which it executes.
3. The Authenticator itself, which provides key management and cryptographic signatures. This may be embedded in the WebAuthn client, or houesd in a separate device entirely. In the latter case, the interface between the WebAuthn client and the authenticator is a separately-defined protocol.

This specification defines the common signature format shared by all the above 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. Refs?
- 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.

§ 4.2.1. Client data used in WebAuthn signatures (dictionary *ClientData*)

The client data represents the contextual bindings of both the WebAuthn Relying Party 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. Its structure is defined by the following Web IDL.

```
dictionary ClientData {
    required DOMString           challenge;
    required DOMString           facet;
    required AlgorithmIdentifier hashAlg;
    JsonWebKey                   tokenBinding;
    WebAuthnExtensions           extensions;
};
```

ArrayBuffer?

The ***challenge*** member contains the base64url encoding of the challenge provided by the RP.

The ***facet*** member contains the fully qualified web origin of the requester, as provided to the authenticator by the client, in the syntax defined by [RFC6454].

The ***hashAlg*** member specifies the hash algorithm used to compute clientDataHash (see §4.2.3 Generating a signature). Use "S256" for SHA-256, "S384" for SHA384, "S512" for SHA512, and "SM3" for SM3 (see §7 IANA Considerations).

The ***tokenBinding*** member contains a JsonWebKey object as defined by Web Cryptography API §JsonWebKey-dictionary describing the public key that this client uses for the Token Binding protocol when communicating with the WebAuthn Relying Party. This can be omitted if no Token Binding has been negotiated between the client and the WebAuthn Relying Party.

The optional ***extensions*** member contains additional parameters generated by processing the extensions passed in by the WebAuthn Relying Party. WebAuthn extensions are detailed in Section §5 WebAuthn Extensions.

§ 4.2.2. 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 of 5 bytes or more, as follows.

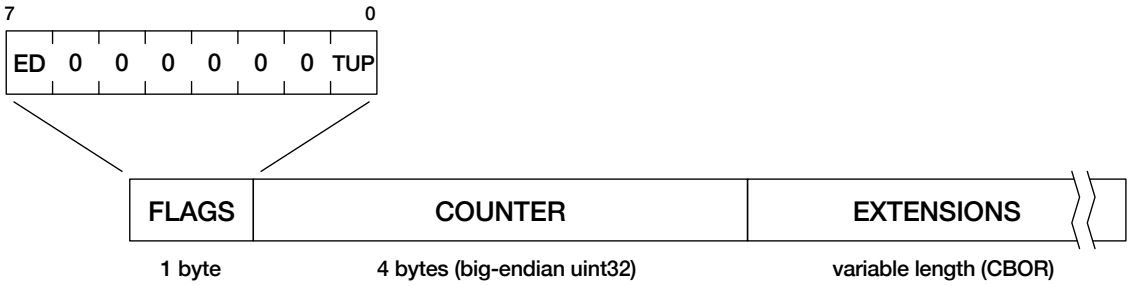
Byte index	Description
0	Flags (bit 0 is the least significant bit): • Bit 0: Test of User Presence (`TUP`) result. • Bits 1-6: Reserved for future use (`RFU`). • Bit 7: Extension data included (`ED`). Indicates if the authenticator data has extensions.
1-4	Signature counter (`signCount`), 32 bit [illegible]
5-	Extension-defined authenticator data. This is a CBOR [RFC7049] map with extension identifiers as keys, and extension authenticator data values as values. See §5 WebAuthn Extensions for details.

first mention of CBOR. Remove, add some clarifying text, or a pointer to the extensions section?

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.

The figure below shows a visual representation of the authenticator data structure.



***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.

§ 4.2.3. Generating a signature

What is the relationship between this and authenticatorMake

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

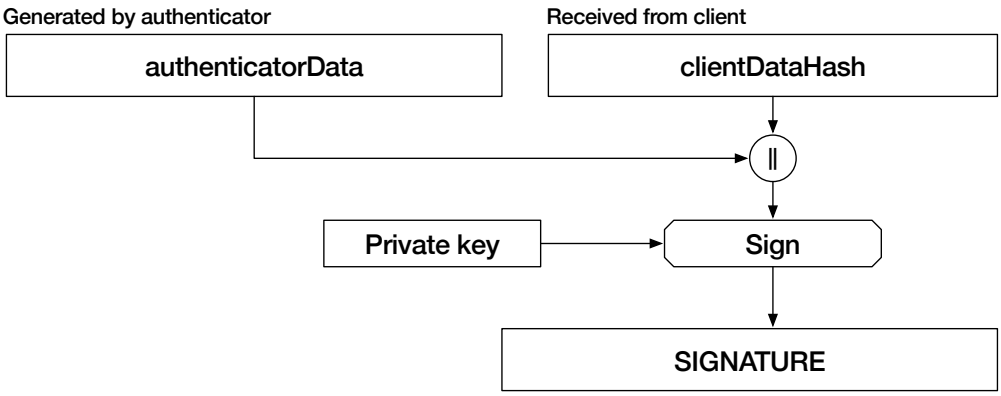
1. Represent the parameters passed in by the RP in the form of a `ClientData` structure.
2. Let ***clientDataJSON*** be the UTF-8 encoded JSON serialization [RFC7159] of this ClientData dictionary.
3. Let ***clientDataHash*** be the hash (computed using hashAlg) of clientDataJSON, as an array.

The *clientDataHash* value is delivered to the authenticator.

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

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*.

There's no mention of incrementing the counter? Also, should authenticatorMakeCredential mention storing / retrieving it?



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. The client, in turn, MUST return clientDataJSON, authenticatorData and the signature to the RP. The clientDataJSON is returned in the `clientData` member of the `WebAuthnAssertion` and `AttestationStatement` structures.

WebAuthnAttestation?

§ 4.3. Credential Attestation Statements

Thinking ahead to future attestation formats, is there a requirement that they MUST sign over clientDataHash?

An attestation statement is a specific type of signature, which contains statements about a credential itself and the authenticator that holds it. Therefore, the procedures for generating attestation statements closely parallel those for generating WebAuthn assertions as described in §4.2 Signature Format, though the semantics of the contextual bindings are quite different.

This specification defines a number of 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. They differ in the details of how the attestation statement is laid out, and how its components are computed. The different attestation types are defined in §4.3.2 Defined Attestation Types.

This specification also defines a number of attestation models. These define how a WebAuthn Relying Party establishes trust in a particular attestation statement, after verifying that it is

cryptographically valid.

Attestation types are orthogonal to attestation models, i.e. attestation types in general are not restricted to a single attestation model. Broadly speaking, attestation types pertain to the syntax of the attestation statement, while attestation models pertain to the semantics.

Maybe a concrete example?

§ 4.3.1. Attestation Models

WebAuthn supports 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 Refs or kill?

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 scoped 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. WebAuthn supports DAA using elliptic curve cryptography and bilinear pairings, called ECDAA (see [FIDOEcdaaAlgorithm]) in this specification.

Compliant servers MUST support all attestation models. Authenticators can choose what attestation model to implement.

> Note: WebAuthn Relying Parties can always decide what attestation models are acceptable to them by policy.

§ 4.3.2. Defined Attestation Types

WebAuthn supports pluggable attestation data types. This allows support of TPM generated attestation data as well as support of other WebAuthn authenticators. As mentioned in §4.3 Credential Attestation Statements, these differ in how the attestation statement is computed and formatted. This section defines these details.

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

§ *4.3.2.1. Packed Attestation (type="packed")*

Packed attestation is a WebAuthn 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).

A Packed Attestation statement has the following format:

```
interface AttestationStatement {
    readonly    attribute unsigned long version;
    readonly    attribute ArrayBuffer   claimedAAGUID;
    readonly    attribute DOMString[]   x5c;
    readonly    attribute DOMString     alg;
    readonly    attribute ArrayBuffer   rawData;
    readonly    attribute ArrayBuffer   signature;
};
```

The ***version*** member specifies the version number of the rawData object. Only version="1" is defined at this time.

The ***claimedAAGUID*** element contains the claimed Authenticator Attestation GUID (a version 4 GUID, see [RFC4122]). This value is used by the WebAuthn Relying Party to look up the trust

anchor for verifying the following signature. 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.

The ***x5c*** attribute contains the attestation certificate and its certificate chain as described in [RFC7515] section 4.1.6.

The ***alg*** element contains the name of the algorithm used to generate the attestation signature according to [RFC7518] section 3.1.

The ***rawData*** object contains the attested public key and the *clientDataHash*. See §4.3.2.1.1 Attestation rawData for details.

The ***signature*** element contains the attestation signature. See §4.3.2.1.2 Signature for details.

§ 4.3.2.1.1. Attestation rawData

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.

The field `rawData` for this type 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-WebAuthn) 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.
2	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 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 §4.2.3 Generating a signature). This is the hash of `clientDataJSON`. The hash algorithm itself is stored in the `clientData` object §4.2 Signature Format.

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 §5 WebAuthn Extensions for a description of the extension mechanism. See §6 Pre-defined extensions 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, it MUST clear the `ED` flag in the third byte.

§ 4.3.2.1.2. Signature

The `signature` is computed over the `rawData` field. The following algorithms must be implemented by servers:

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

Authenticators can choose which algorithm(s) to implement. WebAuthn Relying Parties must implement all the algorithms implemented by the authenticators that they support.

§ 4.3.2.1.3. Packed 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 Extension OID `1 3 6 1 4 1 45724 1 1 4` (id-fido-gen-ce-aaguid) MUST be present, 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].

§ *4.3.2.2. TPM Attestation (type="tpm")*

This attestation type returns an attestation statement in the same format as defined in §4.3.2.1 Packed Attestation (type="packed"). However the `rawData` and `signature` fields are computed differently, as described below.

§ 4.3.2.2.1. Attestation rawData

The value of `rawData` is either a TPM_CERTIFY_INFO or a TPM_CERTIFY_INFO2 structure [TPMv1-2-Part2] sections 11.1 and 11.2, if `version` equals 1. Else, if `version` equals 2, it MUST be a TPMS_ATTEST structure as defined in [TPMv2-Part2] section 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 [[#signature-format]).

§ 4.3.2.2.2. Signature

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

If `version` equals 2, the following algorithms can be used by WebAuthn Authenticators:

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

WebAuthn Relying Parties must implement all the algorithms implemented by the authenticators that they support.

The `signature` is computed over the `rawData` field.

§ 4.3.2.2.3. 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].

§ *4.3.2.3. Android Attestation (type="android")*

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

This type of attestation statement is formatted as follows:

```
interface AndroidAttestation {
    readonly attribute unsigned long version;
    readonly attribute DOMString     safetyNetResponse;
};
```

The ***version*** element is set to the version number of Google Play Services responsible for providing the SafetyNet API.

The ***safetyNetResponse*** element contains the value returned by the above SafetyNet API. This value is a JWS [RFC7515] object (see SafetyNet online documentation) in Compact Serialization.

§ 4.3.2.3.1. Signature

For this attestation type, the ClientData dictionary is extended in the following way:

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

JsonWebKey *publicKey*
: The public key generated by the Authenticator, as a JsonWebKey object (see Web Cryptography API §JsonWebKey-dictionary).

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

DOMString *userAuthentication*
: 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.

optional unsigned long *userAuthenticationValidityDurationSeconds*
: 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.

In order to generate an attestation statement, the client MUST create clientDataJSON by UTF8-encoding a structure of type AndroidAttestationClientData, and compute clientDataHash as the hash of clientDataJSON. It must then provide clientDataHash as the Nonce value when requesting the SafetyNet attestation.

§ 4.3.2.3.2. Verifying AndroidClientData specific contextual bindings

A WebAuthn Relying Party shall verify the clientData contextual bindings (see step 4 in §4.3.3 Verifying an Attestation Statement) as follows:

- Check that `AndroidAttestationClientData.challenge` equals the attestationChallenge that was passed into the `makeCredential()` call.
- Check that the `facet` and `tokenBinding` parameters in the `AndroidAttestationClientData` match the WebAuthn Relying Party App.
- Check that `AndroidAttestationClientData.publicKey` is the same key as the one returned in the `ScopedCredentialInfo` 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.

§ 4.3.3. 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 WebAuthn Relying Party shall:

1. Verify that the attestation statement is properly formatted.
2. If `alg` is not ECDAA (e.g., not "ED256" and not "ED512"):
    - Look up the attestation root certificate from a trusted source. The FIDO Metadata Service [FIDOMetadataService] provides an easy way to access such information. The `claimedAAGUID` can be used for this lookup.
    - Verify that the attestation certificate chain is valid and chains up to a trusted root (following [RFC5280]).
    - Verify that the attestation certificate has the right Extended Key Usage (EKU) based on the WebAuthn Authenticator type (as denoted by the `type` member). In case of a type="tpm", this EKU shall be OID "2.23.133.8.3".
    - If the attestation type is "android", verify that the attestation certificate is issued to the hostname "attest.android.com" (see SafetyNet online documentation).
    - Verify that all issuing CA certificates in the chain are valid and not revoked.
    - Verify the `signature` on `rawData` using the attestation certificate public key and algorithm as identified by `alg`.
    - Verify `rawData` syntax and that it doesn't contradict the signing algorithm specified in `alg`.
    - 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 `claimedAAGUID`. This identifies the Authenticator model.
    - If such extension doesn't exist, the attestation root certificate is dedicated to a single Authenticator model.
3. If `alg` is ECDAA (e.g., "ED256", "ED512"):
    - Look up the DAA root key from a trusted source. The FIDO Metadata Service [FIDOMetadataService] provides an easy way to access such information. The `claimedAAGUID` can be used for this lookup.
    - Perform DAA-Verify on `signature` for `rawData` (see [FIDOEcdaaAlgorithm]).

    - Verify `rawData` syntax and that it doesn't contradict the signing algorithm specified in `alg`.
    - The DAA root key is dedicated to a single Authenticator model.
4. Verify the contextual bindings (e.g., channel binding) from the clientData (see §4.2.3 Generating a signature).
5. Verify that user verification method and other authenticator characteristics related to this authenticator model, match the WebAuthn Relying Party policy. The FIDO Metadata Service [FIDOMetadataService] provides an easy way to access the authenticator characteristics.

The WebAuthn 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 Scoped Credential as one with surrogate basic attestation (see §4.3.1 Attestation Models), if policy allows it. If doing so, there is no cryptographic proof that the Scoped 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 WebAuthn 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.

§ 4.3.4. Security Considerations

§ *4.3.4.1. 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 WebAuthn 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 WebAuthn Authenticator be compromised.
- A WebAuthn Authenticator may be capable of dynamically generating different attestation keys (and requesting related certificates) per origin (following the Privacy CA model). For

example, a WebAuthn 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 WebAuthn Authenticator can implement direct anonymous attestation (see [FIDOEcdaaAlgorithm]). Using this scheme, the authenticator generates a blinded attestation signature. This allows the WebAuthn Relying Party to verify the signature using the DAA root key, but the attestation signature doesn't serve as a global correlation handle.

§ *4.3.4.2. Attestation Certificate and Attestation Certificate CA Compromise*

When an intermediate CA or a root CA used for issuing attestation certificates is compromised, WebAuthn Authenticator attestation keys are still safe although their certificates can no longer be trusted. A WebAuthn 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 WebAuthn Relying Parties must update their trusted root certificates accordingly.

A WebAuthn Authenticator attestation certificate must be revoked by the issuing CA if its key has been compromised. A WebAuthn Authenticator manufacturer may need to ship a firmware update and inject new attestation keys and certificates into already manufactured WebAuthn 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 WebAuthn Authenticators unless the WebAuthn Authenticator manufacturer has this capability.

If attestation certificate validation fails due to a revoked intermediate attestation CA certificate, and the WebAuthn Relying Party's policy requires rejecting the registration/authentication request in these situations, then it is recommended that the WebAuthn Relying Party also un-registers (or marks as "surrogate attestation" (see §4.3.1 Attestation Models), policy permitting) scoped 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 WebAuthn Relying Parties remember intermediate attestation CA certificates during Authenticator registration in order to un-register related Scoped 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 WebAuthn 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.

§ *4.3.4.3. 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 WebAuthn 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 WebAuthn Authenticator device line (i.e., AAGUID), the AAGUID must be specified either in the attestation certificate itself or as an extension in the `rawData`.

§ 5. WebAuthn Extensions

The mechanism for generating scoped credentials, as well as requesting and generating WebAuthn assertions, as defined in §3 Web Authentication API, 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 for registration extension.
- `getAssertion()` request parameters for signature extensions.
- Client processing, and the `ClientData` structure, for registration extensions and signature extensions.
- Authenticator processing, and the authenticatorData structure, for signature extensions.

Or the authn may spontaneously create extensions

When requesting an assertion for a scoped credential, a WebAuthn Relying Party 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.

All WebAuthn extensions are optional for both clients and authenticators. Thus, any extensions requested by a WebAuthn Relying Party may be ignored by the client browser or OS and not passed to the authenticator at all, or they may be ignored by the authenticator. Ignoring an extension is never considered a failure in the WebAuthn API, so when WebAuthn Relying Parties include extensions with any API calls, they must be prepared to handle cases where some or all of those extensions are ignored.

§ 5.1. 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.webauthn.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.

Extensions defined in this specification use a fixed prefix of `webauthn` for the extension identifiers. This prefix should not be used for extensions not defined by the W3C.

§ 5.2. 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 WebAuthn Relying Party would have no indication whether the extension was supported or processed by the client and/or authenticator.

§ 5.3. Extending request parameters

An extension defines two request arguments. The ***client argument*** is passed from the WebAuthn Relying Party 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.

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.

A WebAuthn Relying Party simultaneously requests the use of an extension and sets its client argument by including an entry in the credentialExtensions or assertionExtensions dictionary parameters to the `makeCredential()` or `getAssertion()` 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.webauthn.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.

§ 5.4. 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 WebAuthn Relying Party to verify the processing took place, or if the processing has a result value that the WebAuthn Relying Party 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 client SHOULD add an entry to this dictionary with the extension identifier as the key, and the extension's client data value.

§ 5.5. 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 §4.2.2 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.

§ 5.6. Example extension

This section is not 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.webauthn.geo`. The client argument is the constant value `true`, since the extension does not require the WebAuthn Relying Party to pass any particular information to the client, other than that it requests the use of the extension. The WebAuthn Relying Party sets this value in its request for an assertion:

```
var assertionPromise =
    credentials.getAssertion("SGFuIFNvbG8gc2hvdCBmaXJzdC4",
        {}, /* Empty filter */
        { 'com.example.webauthn.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.webauthn.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
    6C                                          -- Key 1: CBOR text string of 1
        77 65 62 61 75 74 68 6E 2E 67 65 6F     -- "webauthn.geo" UTF-8 string
    82                                          -- Value 1: CBOR array of two e
        FA 42 82 1E B3                          -- Element 1: Latitude as CBOR
        FA C1 5F E3 7F                          -- Element 2: Longitude as CBOI
```

§ 6. Pre-defined extensions

This section defines an initial set of extensions.

§ 6.1. Transaction authorization

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

Extension identifier
`webauthn.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 prompts as well. This allows authenticators without a font rendering engine to be used and also supports a richer visual appearance.

Extension identifier
`webauthn.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.

§ 6.2. Authenticator Selection Extension

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

Extension identifier

`webauthn.authn-sel`

Client argument

A sequence of AAGUIDs:

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

Each AAGUID corresponds to an authenticator attestation that is acceptable to the WebAuthn Relying Party for this credential creation. The list is ordered by decreasing preference.

An AAGUID is defined as an array containing the globally unique identifier of the authenticator attestation being sought.

```
typedef BufferSource AAGUID;
```

Client processing

This extension can only be used during `makeCredential()`. 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.

§ 6.3. AAGUID Extension

Extension identifier

`webauthn.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
>
> - here, or
> - from the attestation certificate (if we have an authenticator specific or authenticator model specific attestation certificate), 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 WebAuthn Relying Party is the DAA root key (i.e., ECPoint2 X, Y). This root key must be dedicated to a single authenticator model.

§ 6.4. SupportedExtensions Extension

Extension identifier

`webauthn.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 SupportedExtensions extension is a list (CBOR array) of extension identifiers encoded as UTF-8 Strings.

§ 6.5. User Verification Index (UVI) Extension

Extension identifier

`webauthn.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 servers to understand whether an authentication was authorized by the exact same biometric data as the initial key generation. This allows the detection and prevention of "friendly fraud".

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

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 WebAuthn packe
81                                              -- TUP and ED set
00 00 00 01                                     -- (initial) signature cour
...                                             -- all public key alg etc.
A1                                              -- extension: CBOR map of c
    6C                                          -- Key 1: CBOR text string
        77 65 62 61 75 74 68 6E 2E 75 76 69     -- "webauthn.uvi" UTF-8 str
    58 20                                       -- Value 1: CBOR byte strin
        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
```

§ 7. 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 [RFC7518].

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
Specification Documents	[FIPS-180-4]
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
Specification Documents	[FIPS-180-4]
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
Specification Documents	[FIPS-180-4]
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
Specification Documents	[OSCCA-SM3]

Algorithm Analysis Document(s)	N/A

§ 8. Sample scenarios

Why aren't these in the use cases section?

This section is not normative.

In this section, we walk through some events in the lifecycle of a scoped 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.

§ 8.1. 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 WebAuthn Relying Party.
2. The WebAuthn 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 WebAuthn Relying Party script. If the user declined to select an authenticator or provide authorization, an appropriate error is returned.
7. If a new credential was created,
    - The WebAuthn Relying Party 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.
    - 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.
    - 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 2

```
var webauthnAPI = window.webauthn;

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

var userAccountInformation = {
    rpDisplayName: "Acme",
    displayName: "John P. Smith",
    name: "johnpsmith@example.com",
    id: "1098237235409872",
    imageURL: "https://pics.acme.com/00/p/aBjjjpqPb.png"
};

// This Relying Party will accept either an ES256 or RS256 credential, bu
// prefers an ES256 credential.
var cryptoParams = [
    {
        type: "ScopedCred",
        algorithm: "ES256"
    },
    {
        type: "ScopedCred",
        algorithm: "RS256"
    }
];
var challenge = "Y2xpbWIgYSBtb3VudGFpbg";
var timeoutSeconds = 300;  // 5 minutes
var blacklist = [];  // No blacklist
var extensions = {"webauthn.location": true};  // Include location inform
                                               // in attestation

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

§ 8.2. 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 WebAuthn 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 WebAuthn 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 WebAuthn Relying Party 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,
    - The script sends the assertion to the server.
    - 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 WebAuthn Relying Party will handle this in its own way.
    - The server now does whatever it would otherwise do upon successful authentication -- return a success page, set authentication cookies, etc.

If the WebAuthn 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 3
var webauthnAPI = window.webauthn;

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

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

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

On the other hand, if the WebAuthn 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 4

```
var webauthnAPI = window.webauthn;

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

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

webauthnAPI.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 appropria
});
```

§ 8.3. 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, the WebAuthn 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 WebAuthn 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.

§ 9. Terminology

Attestation Certificate
: A X.509 Certificate for a key pair used by an Authenticator to attest to its manufacture and capabilities.

Authenticator
: The device used by the user agent to authenticate with the WebAuthn Relying Party. These can be Embedded Authenticators or External Authenticators.

Client
WebAuthn Client
: See Conforming User Agent.

Conforming User Agent
: A user agent which implements algorithms given in this specification, and handles communication between the Authenticator and the WebAuthn Relying Party.

eTLD+1
: The effective top-level domain, plus the first label. Also known as a Registered Domain. See [PSL].

WebAuthn Relying Party
: The entity which needs to rely in the authentication provided by the WebAuthn specification. When registration concludes, the WebAuthn Relying Party has the public key that was created by the Authenticator.

§ 10. Acknowledgements

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

§ Index

§ Terms defined by this specification

§ Terms defined by reference

[HTML] defines the following terms:

Window

[WebIDL-1] defines the following terms:

BufferSource

§ References

§ Normative References

[DOM4]
Anne van Kesteren. DOM Standard. Living Standard. URL: https://dom.spec.whatwg.org/

[FIDOEcdaaAlgorithm]
R. Lindemann; A. Edgington; R. Urian. FIDO ECDAA Algorithm. FIDO Alliance Proposed Standard (To Be Published).

[FIPS-180-4]
FIPS PUB 180-4 Secure Hash Standard. URL: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

[HTML]
Ian Hickson. HTML Standard. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[HTML5]
Ian Hickson; et al. HTML5. 28 October 2014. REC. URL: http://www.w3.org/TR/html5/

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

[RFC4648]

S. Josefsson. The Base16, Base32, and Base64 Data Encodings. October 2006. Proposed Standard. URL: https://tools.ietf.org/html/rfc4648

[RFC7515]

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

[RFC7518]

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

[SEC1]

SEC1: Elliptic Curve Cryptography, Version 2.0. URL: http://www.secg.org/sec1-v2.pdf

[WebCryptoAPI]

Ryan Sleevi; Mark Watson. Web Cryptography API. 11 December 2014. CR. URL: http://www.w3.org/TR/WebCryptoAPI/

[WebIDL-1]

Cameron McCormack; Boris Zbarsky. WebIDL Level 1. 8 March 2016. CR. URL: https://heycam.github.io/webidl/

§ Informative References

[FIDOMetadataService]

R. Lindemann; B. Hill; D. Baghdasaryan. FIDO Metadata Service v1.0. FIDO Alliance Proposed Standard. URL: https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-metadata-service-v1.0-ps-20141208.html

[FIDOSecRef]

R. Lindemann; D. Baghdasaryan; B. Hill. FIDO Security Reference. FIDO Alliance Proposed Standard. URL: https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-security-ref-v1.0-ps-20141208.html

[GeoJSON]

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

[PSL]

Public Suffix List. Mozilla Foundation.

[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

[RFC5280]

D. Cooper; et al. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. May 2008. Proposed Standard. URL: https://tools.ietf.org/html/rfc5280

[RFC6454]

A. Barth. The Web Origin Concept. December 2011. Proposed Standard. URL: https://tools.ietf.org/html/rfc6454

[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

[SECURE-CONTEXTS]

Mike West. Secure Contexts. 26 April 2016. WD. URL: http://www.w3.org/TR/secure-contexts/

[SP800-107r1]

Quynh Dang. NIST Special Publication 800-107: Recommendation for Applications Using Approved Hash Algorithms. August 2012. URL: http://csrc.nist.gov/publications/nistpubs/800-107-rev1/sp800-107-rev1.pdf

[TPMv1-2-Credential-Profiles]

TCG Credential Profiles for TPM Family 1.2. URL: http://www.trustedcomputinggroup.org/wp-content/uploads/Credential_Profiles_V1.2_Level2_Revision8.pdf

[TPMv1-2-Part2]

TPM Main Part 2: TPM Structures. URL: http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Main-Part-2-TPM-Structures_v1.2_rev116_01032011.pdf

[TPMv2-EK-Profile]

TCG EK Credential Profile for TPM Family 2.0. URL: http://www.trustedcomputinggroup.org/wp-content/uploads/Credential_Profile_EK_V2.0_R14_published.pdf

[TPMv2-Part2]

Trusted Platform Module Library, Part 2: Structures. URL: http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-2-Structures-01.16-1.pdf

[UAFProtocol]

R. Lindemann; et al. FIDO UAF Protocol Specification v1.0. FIDO Alliance Proposed Standard. URL: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20150902/fido-uaf-protocol-v1.1-id-20150902.html

§ IDL Index

```
partial interface Window {
    readonly attribute WebAuthentication webauthn;
};

interface WebAuthentication {
    Promise < ScopedCredentialInfo > makeCredential (
        Account                                 accountInformation,
        sequence < ScopedCredentialParameters > cryptoParameters,
        BufferSource                            attestationChallenge,
        optional unsigned long                  credentialTimeoutSeconds
        optional sequence < Credential >        blacklist,
        optional WebAuthnExtensions             credentialExtensions
    );

    Promise < WebAuthnAssertion > getAssertion (
        BufferSource                      assertionChallenge,
        optional unsigned long            assertionTimeoutSeconds,
        optional sequence < Credential > whitelist,
        optional WebAuthnExtensions       assertionExtensions
    );
};

interface ScopedCredentialInfo {
    readonly attribute Credential           credential;
    readonly attribute any                  publicKey;
    readonly attribute WebAuthnAttestation  attestation;
};

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

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

interface WebAuthnAssertion {
    readonly attribute Credential  credential;
```

```
    readonly attribute ArrayBuffer clientData;
    readonly attribute ArrayBuffer authenticatorData;
    readonly attribute ArrayBuffer signature;
};

dictionary WebAuthnExtensions {
};

interface WebAuthnAttestation {
    readonly    attribute DOMString     type;
    readonly    attribute ArrayBuffer   clientData;
    readonly    attribute any           statement;
};

enum CredentialType {
    "ScopedCred"
};

interface Credential {
    readonly attribute CredentialType type;
    readonly attribute BufferSource   id;
};

dictionary ClientData {
    required DOMString           challenge;
    required DOMString           facet;
    required AlgorithmIdentifier hashAlg;
    JsonWebKey                   tokenBinding;
    WebAuthnExtensions           extensions;
};

interface AttestationStatement {
    readonly    attribute unsigned long version;
    readonly    attribute ArrayBuffer   claimedAAGUID;
    readonly    attribute DOMString[]   x5c;
    readonly    attribute DOMString     alg;
    readonly    attribute ArrayBuffer   rawData;
    readonly    attribute ArrayBuffer   signature;
};

interface AndroidAttestation {
    readonly attribute unsigned long version;
    readonly attribute DOMString     safetyNetResponse;
```

```
};

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

typedef sequence < AAGUID > AuthenticatorSelectionList;

typedef BufferSource AAGUID;
```