Abstract

This specification defines an API enabling the creation and use of strong, attested, cryptographic scoped credentials by web applications, for the purpose of strongly authenticating users. Conceptually, authenticators are responsible for ensuring that no operation is performed without user consent. Authenticators provide cryptographic proof of their properties to relying parties via attestation. This specification also describes the functional model for WebAuthn conformant authenticators, including their 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

Abstract

This specification defines an API enabling the creation and use of strong, attested, scoped, public key-based credentials by web applications, for the purpose of strongly authenticating users. Conceptually, one or more credentials, each scoped to a given Relying Party, are created and stored on an authenticator by the user agent in conjunction with the web application. The user agent mediates access to public key credentials in order to preserve user privacy. Authenticators are responsible for ensuring that no operation is performed without user consent. Authenticators provide cryptographic proof of their properties to relying parties via attestation. This specification also describes the functional model for WebAuthn conformant authenticators, including their 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
This section is not normative.

This section is not normative.
This specification defines an API enabling the creation and use of strong, attested, cryptographic scoped credentials by web applications, for the purpose of strongly authenticating users. A scoped credential is created and stored by an authenticator at the behest of a Relying Party, subject to user consent. Subsequently, the scoped credential can only be accessed by origins belonging to that Relying Party. This scoping is enforced jointly by conforming User Agents and authenticators. Additionally, privacy across Relying Parties is maintained; Relying Parties are not able to detect any properties, or even the existence, of credentials scoped to other Relying Parties.

Relying Parties employ the Web Authentication API during two distinct, but related, ceremonies involving a user. The first is Registration, where a scoped credential is created on an authenticator, and associated by a Relying Party with the present user’s account (the account may already exist or may be created at this time). The second is Authentication, where the Relying Party is presented with an Authentication Assertion proving the presence and consent of the user who registered the scoped credential. Functionally, the Web Authentication API comprises two methods (along with associated data structures): makeCredential() and getAssertion(). The former is used during Registration and the latter during Authentication.

Broadly, compliant authenticators protect scoped credentials, and interact with user agents to implement the Web Authentication API. Some authenticators may run on 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 Execution Environment (TEE) applet, a Trusted Platform Module (TPM), or a Secure Element (SE) integrated into the computing device in conjunction with some means for user interaction, along with appropriate platform software to mediate access to these components’ functionality. Other authenticators may operate autonomously from the computing device running the user agent, and be accessed over a transport such as Universal Serial Bus (USB), Bluetooth Low Energy (BLE) or Near Field Communications (NFC).

1.1. Use Cases

The below use case scenarios illustrate use of two very different types of authenticators, as well as outline further scenarios. Additional scenarios, including sample code, are given later in 11 Sample scenarios.

1.1.1. Registration

* On a user navigates to example.com in a 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."

1.1.2. Authentication

* On a laptop or desktop:
  * User navigates to example.com in a browser, sees an option to "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 their phone:
  * User sees a discrete prompt or notification, "Sign in to
This specification describes the syntax of all CBOR-encoded data using the CBOR Data Definition Language (CDDL) [CDDL].

1.1.3. Other use cases and configurations

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

* A user navigates to example.com on their laptop, is guided through a flow to create and register a credential on their phone.

* A user obtains an discrete, roaming authenticator, such as a "fob" with USB or USB+NFC/BLE connectivity options, loads example.com in their browser on a laptop or phone, and is guided through a flow to create and register a credential on the fob.

* A Relying Party prompts the user for their authorization gesture in order 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. Conforming 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 conformant User Agent MUST also be a conformant 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 conformant authenticator (see 5 WebAuthn Authenticator model). This is a set of functional and security requirements for an authenticator to be usable by a Conforming User Agent. As described in 1.1 Use Cases, an authenticator 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:

- Base64url encoding
- CBOR
- A number of structures in this specification, including attestation statements and extensions, are encoded using the Compact Binary Object Representation (CBOR) [RFC7049].
- CDDL

This specification describes the syntax of all CBOR-encoded data using the CBOR Data Definition Language (CDDL) [CDDL].

The API described in this document is an extension of the Credential concept defined in [CREDENTIAL-MANAGEMENT-1].
DOMException and the DOMException values used in this specification are defined in [DOM4].

**HTML**

The concepts of current settings object, origin, opaque origin, relaxing the same-origin restriction, and the Navigator interface are defined in [HTML51].

**Web Cryptography API**

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

**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. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. Terminology

**ASCII case-insensitive match**

A method of testing two strings for equality by comparing them exactly, code point for code point, except that the codepoints in the range U+0041 - U+005A (i.e., LATIN CAPITAL LETTER A to LATIN CAPITAL LETTER Z) and corresponding codepoints in the range U+0061 - U+007A (i.e., LATIN SMALL LETTER A to LATIN SMALL LETTER Z) are also considered to match.

**Assertion**

See Authentication Assertion.

**Attestation**

Generally, a statement that serves to bear witness, confirm, or authenticate. In the WebAuthn context, attestation is employed to attest to the provenance of an authenticator and the data it emits; including, for example: credential IDs, credential key pairs, signature counters, etc. Attestation information is conveyed in attestation objects. See also attestation statement format, and attestation type.

**Attestation Certificate**

A X.509 Certificate for the attestation key pair used by an Authenticator to attest to its manufacture and capabilities. At registration time, the authenticator uses the attestation private key to sign the Relying Party-specific credential public key (and additional data) that it generates and returns via the authenticatorMakeCredential operation. Relying Parties use the attestation public key conveyed in the attestation certificate to verify the attestation signature. Note that in the case of self attestation, the authenticator has no distinct attestation key pair or attestation certificate, see self attestation for details.

**Authentication**

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

**ECMAScript**

%ArrayBuffer% is defined in [ECMAScript].

**HTML**

The concepts of relevant settings object, origin, opaque origin, and is a registrable domain suffix of or is equal to are defined in [HTML52].

**Web Cryptography API**

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

**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. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. Terminology

**Assertion**

See Authentication Assertion.

**Attestation**

Generally, a statement that serves to bear witness, confirm, or authenticate. In the WebAuthn context, attestation is employed to attest to the provenance of an authenticator and the data it emits; including, for example: credential IDs, credential key pairs, signature counters, etc. Attestation information is conveyed in attestation objects. See also attestation statement format, and attestation type.

**Attestation Certificate**

A X.509 Certificate for the attestation key pair used by an authenticator to attest to its manufacture and capabilities. At registration time, the authenticator uses the attestation private key to sign the Relying Party-specific credential public key (and additional data) that it generates and returns via the authenticatorMakeCredential operation. Relying Parties use the attestation public key conveyed in the attestation certificate to verify the attestation signature. Note that in the case of self attestation, the authenticator has no distinct attestation key pair or attestation certificate, see self attestation for details.

**Authentication**
The ceremony where a user, and the user's computing device(s) (containing at least one authenticator) work in concert to cryptographically prove to an Relying Party that the user controls the private key associated with a previously-registered scoped credential (see Registration). Note that this includes employing user verification.

Authentication Assertion

The cryptographically signed AuthenticationAssertion object returned by an authenticator as the result of a authenticatorGetAssertion operation.

Authenticator

A cryptographic device used by a WebAuthn Client to (i) generate a public key credential and register it with a Relying Party, and (ii) subsequently used to cryptographically sign and return, in the form of an AuthenticationAssertion, a challenge and other data presented by a Relying Party (in concert with the WebAuthn Client) in order to effect authentication.

Authorization Gesture

Essentially the same as user verification.

Ceremony

The concept of a ceremony [Ceremony] is an extension of the concept of a network protocol, with human nodes alongside computer nodes and with communication links that include UI, human-to-human communication and transfers of physical objects that carry data. What is out-of-band to a ceremony is in-band to a protocol. In this specification, Registration, Authentication, and user verification are ceremonies.

Client

See Conforming User Agent.

Conforming User Agent

A user agent implementing, in conjunction with the underlying platform, the Web Authentication API and algorithms given in this specification, and handling communication between Authenticators and Relying Parties.

Credential Public Key

The public key portion of a Relying Party-specific credential key pair, generated by an authenticator and returned to an Relying Party at registration time (see also scoped credential). The private key portion of the credential key pair is known as the credential private key. Note that in the case of self attestation, the credential key pair is also used as the attestation key pair, see self attestation for details.

Registration

A user with an authenticator as part of a ceremony, such as registration or authentication. By making such an authorization gesture, a user provides consent for (i.e., authorizes) a ceremony to proceed. This may involve user verification if the employed authenticator is capable, or it may involve a simple test of user presence.

Biometric Recognition

The automated recognition of individuals based on their biological and behavioral characteristics [ISOBiometricVocabulary].

Ceremony

The concept of a ceremony [Ceremony] is an extension of the concept of a network protocol, with human nodes alongside computer nodes and with communication links that include user interface(s), human-to-human communication, and transfers of physical objects that carry data. What is out-of-band to a protocol is in-band to a ceremony. In this specification, Registration and Authentication are ceremonies, and an authorization gesture is often a component of those ceremonies.

Client

See Conforming User Agent.

Conforming User Agent

A user agent implementing, in conjunction with the underlying platform, the Web Authentication API and algorithms given in this specification, and handling communication between Authenticators and Relying Parties.

Credential Public Key

The public key portion of a Relying Party-specific credential key pair, generated by an authenticator and returned to an Relying Party at registration time (see also public key credential). The private key portion of the credential key pair is known as the credential private key. Note that in the case of self attestation, the credential key pair is also used as the attestation key pair, see self attestation for details.
The ceremony where a user, a Relying Party, and the user's computing device(s) (containing at least one authenticator) work together to create a new credential and associate it with the user's Relying Party account. Note that this includes employing user verification.

Relying Party

The entity whose web application utilizes the Web Authentication API to register and authenticate users. See Registration and Authentication, respectively.

Note: While the term Relying Party is used in other contexts (e.g., X.509 and OAuth), an entity acting as a Relying Party in one context is not necessarily a Relying Party in other contexts.

Relying Party Identifier

RP ID

An identifier for the Relying Party on whose behalf a given registration or authentication ceremony is being performed. Scoped credentials can only be used for authentication by the same entity (as identified by RP ID) that created the credential, and the RP ID for a WebAuthn operation is set to the current settings object's origin. This default can be overridden by the caller subject to certain restrictions, as specified in 4.1.1 Create a new credential - makeCredential() method and 4.1.2 Use an existing credential - getAssertion() method.

Scoped Credential

Generically, a credential is data one entity presents to another in order to authenticate the former's identity [RFC4949]. A WebAuthn scoped credential is a \{(identifier, type)\} pair identifying authentication information established by the authenticator and the Relying Party, together, at registration time. The authentication information consists of an asymmetric key pair, where the public key portion is returned to the Relying Party, which stores it in conjunction with the present user's account. The authenticator maps the private key to the Relying Party's RP ID and stores it. Subsequently, only that Relying Party, as identified by its RP ID, is able to employ the scoped credential in authentication ceremonies, via the getAssertion() method. The Relying Party uses its copy of the stored public key to verify the resultant Authentication Assertion.

User Consent

Test of User Presence

TUP

A test of user presence is a simple form of authorization gesture and technical process where a user interacts with an authenticator by (typically) simply touching it (other modalities may also exist), yielding a boolean result. Note that this does not constitute user verification because TUP, by definition, is not capable of biometric recognition, nor does it involve the presentation of a shared secret such as a password or PIN.

Client-side-resident Credential Private Key

A Client-side-resident Credential Private Key is stored either on the client platform, or in some cases on the authenticator itself, e.g., in the case of a discrete first-factor roaming authenticator. Such client-side credential private key storage...
management actions such as credential deletion are considered to be the state of the authenticator. In other words, such an interface is to reset the authenticator to a clean state or to inspect the current interface for management. Such an interface may be used, for example, through a touch plus pin code, password entry, or other modality. Note that invocation of said operations implies use of key material managed by the authenticator.

User Verification

The process by which an authenticator locally authorizes the invocation of the authenticatorMakeCredential and authenticatorGetAssertion operations, for example through a touch plus pin code, a password, a gesture (e.g., presenting a fingerprint), or other modality. Note that invocation of said operations implies use of key material managed by the authenticator.

User Consent

User consent means the user agrees with what they are being asked, i.e., it encompasses reading and understanding prompts. An authorization gesture is a ceremony component often employed to indicate user consent.

WebAuthn Client

See Conforming User Agent.

4. Web Authentication API

This section normatively specifies the API for creating and using scoped credentials. The basic idea is that the credentials belong to the user and are managed by an authenticator, with which the 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 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, interacting with 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.

In addition to the above script interface, the authenticator may implement (or come with client software that implements) a user interface for management. Such an interface may be used, for example, to reset the authenticator to a clean state or to inspect the current state of the authenticator. In other words, such an interface is similar to the user interfaces provided by browsers for managing user state such as history, saved passwords and cookies. Authentication management actions such as credential deletion are considered to be the responsibility of such a user interface and are deliberately omitted from the API exposed to scripts.

User Verification

The technical process by which an authenticator locally authorizes the invocation of the authenticatorMakeCredential and authenticatorGetAssertion operations. User verification may be instigated through various authorization gesture modalities; for example, through a touch plus pin code, password entry, or biometric recognition (e.g., presenting a fingerprint).

User Consent

User consent means the user agrees with what they are being asked, i.e., it encompasses reading and understanding prompts. An authorization gesture is a ceremony component often employed to indicate user consent.

WebAuthn Client

Also referred to herein as simply a client. See also Conforming User Agent.

4. Web Authentication API

This section normatively specifies the API for creating and using public key credentials. The basic idea is that the credentials belong to the user and are managed by an authenticator, with which the 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 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, interacting with 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.

In addition to the above script interface, the authenticator may implement (or come with client software that implements) a user interface for management. Such an interface may be used, for example, to reset the authenticator to a clean state or to inspect the current state of the authenticator. In other words, such an interface is similar to the user interfaces provided by browsers for managing user state such as history, saved passwords and cookies. Authentication management actions such as credential deletion are considered to be the responsibility of such a user interface and are deliberately omitted from the API exposed to scripts.
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 origin, and cannot be replayed against a different origin, by incorporating the origin in its responses. Specifically, as defined in 5.2 Authenticator operations, the full origin of the requester is included, and signed over, in the attestation object 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 Relying Parties from probing for the presence of credentials belonging to other 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 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 Relying Party maintains multiple origins.

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

The Web Authentication API is defined by the union of the Web IDL fragments presented in the following sections. A combined IDL listing is given in the IDL Index. The API is defined as a part of the Navigator interface:

```
interface PartialNavigator {
  optional attribute WebAuthentication authentication;
}
```

4.1. WebAuthentication Interface

```
interface WebAuthentication {
  Promise<ScopedCredentialInfo> makeCredential(
    Account accountInformation,
    sequence<ScopedCredentialParameters> cryptoParameters,
    optional ScopedCredentialOptions options
  );

  Promise<AuthenticationAssertion> getAssertion(
    BufferSource assertionChallenge,
    optional AssertionOptions options
  );
}
```

4.1. PublicKeyCredential Interface

```
interface PublicKeyCredential : Credential {
  readonly attribute ArrayBuffer rawId;
  optional attribute AuthenticationResponse response;
  readonly attribute AuthenticationExtensions clientExtensionResults;
}
```

The PublicKeyCredential interface inherits from Credential [CREDENTIAL-MANAGEMENT-1], and contains the attributes that are returned to the caller when a new credential is created, or a new assertion is requested.

```
interface PublicKeyCredential : SharedCredential {
  readonly attribute ArrayBuffer base64Credential;
}
```

This attribute returns the ArrayBuffer contained in the [[identifier]] internal slot.

```
interface PublicKeyCredential {
  public get PublicKeyCredential();
}
```

This attribute contains the Authenticator's response to the client's request to either create a public key credential, or generate an authentication assertion. If the PublicKeyCredential is created in response to create(), this attribute's value will be an AuthenticationResponse; otherwise, the PublicKeyCredential was created in response to get(), and this attribute's value will be an AuthenticationResponse.

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 origin, and cannot be replayed against a different origin, by incorporating the origin in its responses. Specifically, as defined in 5.2 Authenticator operations, the full origin of the requester is included, and signed over, in the attestation object 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 Relying Parties from probing for the presence of credentials belonging to other 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 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 Relying Party maintains multiple origins.

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

The Web Authentication API is defined by the union of the Web IDL fragments presented in the following sections. A combined IDL listing is given in the IDL Index.
not result in the older credential being offered to the user. Any calls to getAssertion() that do not specify allowList will result in the promise being rejected with a new error. Successful execution of this method: if the credential is not listed in the excludeList member of options, then after successful execution of this method:

```

* 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. An authenticator is only required to store one credential for any given value of accountInformation. Specifically, if an authenticator already has a credential for the specified value of id in accountInformation, and if this credential is not listed in the excludeList member of options, then after successful execution of this method:

```

```

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

4.1.1. Create a new credential - makeCredential() method

With this method, a script can request the User Agent to create a new credential of a given type and persist it to the underlying platform, which may involve data storage managed by the browser or the OS. The user agent will prompt the user to approve this operation. On success, the promise will be resolved with a 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. An authenticator is only required to store one credential for any given value of accountInformation. Specifically, if an authenticator already has a credential for the specified value of id in accountInformation, and if this credential is not listed in the excludeList member of options, then after successful execution of this method:
  + Any calls to getAssertion() that do not specify allowList will not result in the older credential being offered to the user;
  + Any calls to getAssertion() that specify the older credential will not result in the older credential being offered to the user.

```
5. If normalizedAlgorithm is empty and cryptoParameters was not empty, + Add a new object of type ScopedCredentialParameters to + Add a new object of type ScopedCredentialParameters to + Add a new object of type ScopedCredentialParameters to + Add a new object of type ScopedCredentialParameters to procedure, then stop processing current and move on to the + Let normalizedAlgorithm be the result of normalizing an + Let normalizedAlgorithm be the result of normalizing an + Let normalizedAlgorithm be the result of normalizing an + Let normalizedAlgorithm be the result of normalizing an algorithm. When this method is invoked, the user agent MUST execute the following algorithm: 1. If the timeout member of options is present, 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 adjustableTimeout to this adjusted value. If the timeout was not specified, then set adjustableTimeout to a platform-specific default.

2. Let promise be a new Promise. Return promise and start a timer for adjustableTimeout milliseconds. Then asynchronously continue executing the following steps, if any fatal error is encountered in this process other than the ones enumerated below, cancel the timer, reject promise with a DOMException whose name is "UnknownError", and terminate this algorithm.

3. Set callerOrigin to the current settings object's origin. If callerOrigin is an opaque origin, reject promise with a DOMException whose name is "NotAllowedError", and terminate this algorithm. Otherwise:
   - If the rpId member of options is not present, then set rpId to callerOrigin.
   - If the rpId member of options is present, then invoke the procedure used for relaxing the same-origin restriction by setting the document.domain attribute, using rpId as the given value but without changing the current document's domain. If no errors are thrown, set rpId to the value of host as returned by this procedure. Otherwise, reject promise with a DOMException whose name is "SecurityError", and terminate this algorithm.

4. Process each element of cryptoParameters using the following steps, to produce a new sequence normalizedParameters.
   - If current.type does not contain a ScopedCredentialType 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 [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.
   - If normalizedAlgorithm is empty and cryptoParameters was not empty, cancel the timer started in step 2, reject promise with a DOMException whose name is "SecurityError", and terminate this algorithm.

5. If normalizedAlgorithm is empty and cryptoParameters was not empty, return a DOMException whose name is "NotSupportedError", and terminate this algorithm.

6. Let global be the PublicKeyCredential interface object's global. Let options.user, the displayName member of options.user, or the id member of options.user be not present, return a TypeError simple exception.

7. If the id member of options is present, set id to options.user, otherwise, set id to the current settings object's origin.

8. Let normalizedParameters be a new list whose items are pairs of [PublicKeyCredentialType] and a dictionary type (as returned by navigator.credentials.create()) which contains 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.

9. For each current of options.parameters:
   - Let current be the currently selected element of cryptoParameters.
   - Let normalizedAlgorithm be the result of normalizing an algorithm [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.
   - If normalizedAlgorithm is empty and cryptoParameters was not empty, cancel the timer started in step 2, reject promise with a DOMException whose name is "SecurityError", and terminate this algorithm.

10. If normalizedParameters is empty and options.parameters is not present, return a DOMException whose name is "NotSupportedError", and terminate this algorithm.

11. Let clientExtensions be a new map and let authenticatorExtensions be a new map, each mapping a string to a value. For each extension in clientExtensions:
   - If extension is not present in options.parameters, reject promise with a DOMException whose name is "NotSupportedError", and terminate this algorithm.

12. For each extension in authenticatorExtensions:
   - If extension is not present in options.parameters, reject promise with a DOMException whose name is "NotSupportedError", and terminate this algorithm.

13. If the rpId member of options is not present, then set rpId to callerOrigin.

14. Otherwise:
   - Let effectiveDomain be the callerOrigin's effective domain.
   - Otherwise:
     - If current.domain is not present, then set current.domain to effectiveDomain.
     - Let callerOrigin be the origin specified by this PublicKeyCredential interface object's relevant settings object. If callerOrigin is an opaque origin, return a DOMException whose name is "NotAllowedError", and terminate this algorithm.
     - If the id member of options is present, set rpId to options.user, otherwise, set rpId to the current settings object's origin.
     - Let options be a new options parameter.
     - Let options.name be a String whose value is "publicKey".
     - Let options.polymorphicKey be true.
     - Let options.cryptoParameters be the options specified by the PublicKeyCredential interface object.
     - Let options.id be the options.id.
     - Let optionsGENERATIONDATA be the options.GENERATIONDATA.
     - Let options.clientExtensionData be the options.clientExtensionData.
     - Let options.clientExtensionData be the options.clientExtensionData.
     - Let options.clientExtensionData be the options.clientExtensionData.
     - Let options.clientExtensionData be the options.clientExtensionData.

15. Note: This algorithm is synchronous; the Promise resolution/rejection is taken care of by navigator.credentials.create(). This method accepts a single argument:

   options
   - This argument is a CredentialCreationOptions object whose options["publicKey"] member contains a MakeCredentialOptions object specifying how the credential is to be made.

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

1. Assert: options["publicKey"] is present.

2. Let options be the value of options["publicKey"].

3. If any of the name member of options.rp, the name member of options.user, the displayName member of options.user, or the id member of options.user are not present, return a TypeError simple exception.

4. If the timeout member of options is present, 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 adjustableTimeout to this adjusted value. If the timeout member of options is not present, then set adjustableTimeout to a platform-specific default.

5. Let global be the PublicKeyCredential interface object's environment settings object's global object.

6. Let callerOrigin be the origin specified by this PublicKeyCredential interface object's relevant settings object. If callerOrigin is an opaque origin, return a DOMException whose name is "NotAllowedError", and terminate this algorithm.

7. If the id member of options is not present, then set rpId to callerOrigin.

8. Let normalizedParameters be a new list whose items are pairs of PublicKeyCredentialType and a dictionary type (as returned by navigator.credentials.create()) which contains 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.

9. For each current of options.parameters:
   - Let current be the currently selected element of cryptoParameters.
   - Let normalizedAlgorithm be the result of normalizing an algorithm [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.
   - If normalizedAlgorithm is empty and cryptoParameters was not empty, cancel the timer started in step 2, reject promise with a DOMException whose name is "SecurityError", and terminate this algorithm.

10. If normalizedParameters is empty and options.parameters is not present, return a DOMException whose name is "NotSupportedError", and terminate this algorithm.

11. Let clientExtensions be a new map and let authenticatorExtensions be a new map, each mapping a string to a value. For each extension in client Extensions: - If extension is not present in options.parameters, reject promise with a DOMException whose name is "NotSupportedError", and terminate this algorithm.
11. While issuedRequests is not empty, perform the following actions.

6. If the extensions member of options is present, process any extensions supported by this client platform, to produce the extension data that needs to be sent to the authenticator. If an error is encountered while processing an extension, skip that extension and do not produce any extension data for it. Call the result of this processing clientExtensions.

7. Use attestationChallenge, callerOrigin and rpId, along with the token binding key associated with callerOrigin (if any), to create a ClientData structure representing this request. Choose a hash algorithm for hashAlg and compute the clientDataJSON and its tokenBinding.

8. Initialize issuedRequests and currentlyAvailableAuthenticators to empty lists.

9. For each authenticator currently available on this platform, add the authenticator to currentlyAvailableAuthenticators unless the attachment member of options is present. In that case, let attachment be attachment, and add the authenticator to currentlyAvailableAuthenticators if its attachment modality matches attachment.

10. For each authenticator in currentlyAvailableAuthenticators: asynchronously invoke the authenticatorMakeCredential operation on that authenticator with rpId, clientDataHash, accountInformation, normalizedParameters, excludeList and clientExtensions as parameters. Add a corresponding entry to issuedRequests.

+ For each credential C in the excludeList member of options that has a non-empty transports list, optionally use only the specified transports to test for the existence of C.

11. While issuedRequests is not empty, perform the following actions.

   1. If C.transports is not empty, and authenticator is not capable of storing a Client-Side-Resident Credential represented by clientDataJSON. continue.

   2. If requireResidentKey is set to true and the authenticator is not a registration extension, then continue.

   3. If extensionId is not an authenticator extension, then continue.

   4. Let authenticatorExtensionInput be the (CBOR) result of running extensionId’s client extension processing algorithm on clientExtensionInput. If the algorithm returned an error, continue.

   5. Set authenticatorExtensions[extensionId] to the base64url encoding of authenticatorExtensionInput.

   6. If the extensions member of options is present, then for each extensionId -> clientExtensionInput of options.extensions:

      1. If extensionId is not supported by this client platform or is not a registration extension, then continue.

      2. Set clientExtensions[extensionId] to clientExtensionInput.

      3. If extensionId is not an authenticator extension, then continue.

   7. Let selectedAuthenticators be a new ordered set consisting of all authenticators available on this platform.

   8. For each authenticator currently available on this platform, add the authenticator to selectedAuthenticators if its attachment modality matches attachment.

   9. Let selectedAuthenticators be a new ordered set consisting of all authenticators available on this platform.

   10. Let clientDataJSON be the JSON-serialized client data constructed from collectedClientData.

   11. Let clientDataHash be the hash of the serialized client data represented by clientDataJSON.

12. If the extensions member of options is present, then for each extensionId -> clientExtensionInput of options.extensions:

      1. If extensionId is not supported by this client platform or is not a registration extension, then continue.

      2. Set clientExtensions[extensionId] to clientExtensionInput.

      3. If extensionId is not an authenticator extension, then continue.

   13. Let collectedClientData be a new CollectedClientData instance whose fields are:

      * challenge: The base64url encoding of options.challenge
      * origin: The unicode serialization of rpId
      * hashAlg: The recognized algorithm name of the hash algorithm selected by the client for generating the hash of the serialized client data
      * tokenBinding: The Token Binding ID associated with callerOrigin, if one is available.

14. Let clientExtensions be a new map.

15. Let clientExtensions be a new map.

16. Let currentlyAvailableAuthenticators be a new ordered set consisting of all authenticators available on this platform.

17. Let selectedAuthenticators be a new ordered set.

18. If currentlyAvailableAuthenticators is empty, return a DOMException whose name is "NotSupportedError", and terminate this algorithm.

19. If options.authenticationSelection is present, iterate through currentlyAvailableAuthenticators and do the following for each authenticator:

      1. If attachment is present and its value is not equal to authenticator’s attachment modality, continue.

      2. If requireResidentKey is set to true and the authenticator is not capable of storing a Client-Side-Resident Credential Private Key, continue.

      3. Append authenticator to selectedAuthenticators.

20. If selectedAuthenticators is empty, return a DOMException whose name is "ConstraintError", and terminate this algorithm.

21. Let issuedRequests be a new ordered set.

22. For each authenticator in currentlyAvailableAuthenticators:

      1. Let excludeList be a new list.

      2. For each credential C in options.excludeList:

         1. If C.transports is not empty, and authenticator is connected over a transport not mentioned in C.transports, the client MAY continue.

         2. Otherwise, Append C to excludeList.

      3. In parallel, invoke the authenticatorMakeCredential operation on authenticator with rpId, clientDataHash, options, rp, options.user, normalizedParameters, excludeList, and authenticatorExtensions as parameters.
During the above process, the user agent SHOULD show some UI to the user, and terminate this algorithm.

4. Reject promise with a DOMException whose name is "NotAllowedError", and terminate this algorithm.

5. For each remaining authenticator in issuedRequests invoke the authenticatorCancel operation on that authenticator and remove its entry from the list.

6. Return a DOMException whose name is "NotAllowedError".

7. If any authenticator returns an error status, remove authenticator from issuedRequests.

8. If any authenticator indicates success, remove authenticator from issuedRequests.

9. If any authenticator indicates success, remove authenticator from issuedRequests.

During the above process, the user agent SHOULD show some UI to the user, and terminate this algorithm.
6. Initialize issuedRequests to an empty list.

5. Use assertionChallenge, callerOrigin and rpId, along with the token result of this processing clientExtensions.

error is encountered while processing an extension, skip that extension data that needs to be sent to the authenticator. If an error is encountered, set rpId to the value of host as value but without changing the current document's domain. If setting the document.domain attribute, using rpId as the given origin, return DOMException whose name is "NotAllowedError", and terminate this algorithm.

1. If the timeout member of options is present, 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 timeout was not specified, set adjustedTimeout to a platform-specific default.

3. Set callerOrigin to the current settings object's origin. If callerOrigin is an opaque origin, reject promise with a DOMException whose name is "UnknownError", and terminate this algorithm.

3. If extensionId is not supported by this client platform or is not an authentication extension, then continue.

1. If extensionId is not an authenticator extension, then continue.

2. If effectiveDomain is null, then return a DOMException whose name is "SecurityError" and terminate this algorithm.

3. If rpId is not a registrarDom suffix of and is not equal to effectiveDomain, return a DOMException whose name is "SecurityError", and terminate this algorithm.

4. Set rpId to the rpd.

6. Let clientExtensions be a new map and let authenticationExtensions be a new map.

7. If the extensions member of publicKeyOptions is not present, then set clientExtensions for this method.

4. If the extensions member of options is present, process any extensions supported by this client platform, to produce the extension data that needs to be sent to the authenticator. If an error is encountered while processing an extension, skip that extension and do not produce any clientExtensions data for it. Call the result of this processing clientExtensions.

5. Use assertionChallenge, callerOrigin and rpId, along with the token binding key associated with callerOrigin (if any), to create a ClientData structure representing this request. Choose a hash algorithm for hashAlg and compute the clientExtensions data hash.

6. Initialize issuedRequests to an empty list.

7. For each authenticator currently available on this platform,
perform the following steps:
+ If the allowList member of options is empty, let credentialList be an empty list. Otherwise, execute a platform-specific procedure to determine which, if any, credentials listed in allowList might be present on this authenticator, and set credentialList to this filtered list. If no such filtering is possible, set credentialList to an empty list.
+ For each credential C within the credentialList that has a non-empty transports list, optionally use only the specified transports to get assertions using credential C.
+ If the above filtering process concludes that none of the credentials on the allowList can possibly be on this authenticator, do not perform any of the following steps for this authenticator, and proceed to the next authenticator (if any).
+ Asynchronously invoke the authenticatorGetAssertion operation on this authenticator with rpId, clientDataHash, credentialList, and clientExtensions as parameters.
+ Add an entry to issuedRequests, corresponding to this request.

8. 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:
  o Remove this authenticator’s entry from issuedRequests.
  o Create a new AuthenticationAssertion object named value and populate its fields with the values returned from the

9. Let clientDataJSON be the JSON-serialized client data constructed from collectedClientData.
10. Let clientDataHash be the hash of the serialized client data represented by clientDataJSON.
11. Let issuedRequests be a new ordered set.
12. If there are no authenticators currently available on this platform, return a DOMException whose name is "NotAllowedError", and terminate this algorithm.
13. For each authenticator currently available on this platform, perform the following steps:
1. Let credentialList be a new list.
2. If allowList is not empty, execute a platform-specific procedure to determine which, if any, credentials in publicKeyOptions.allowList are present on this authenticator by matching with publicKeyOptions.allowList.id and publicKeyOptions.allowList.type, and set credentialList to this filtered list.
3. If credentialList is empty then continue.
4. In parallel, for each credential C in credentialList:
   1. If C.transports is not empty, the client SHOULD select one transport from transports. Then, using transport, invoke the authenticatorGetAssertion operation on authenticator with rpId, clientDataHash, credentialList, and authenticatorExtensions as parameters.
   2. Otherwise, using local configuration knowledge of the appropriate transport to use with authenticator, invoke the authenticatorGetAssertion operation on authenticator with rpId, clientDataHash, credentialList, and clientExtensions as parameters.
5. Append authenticator to issuedRequests.
14. Start a timer for adjustedTimeout milliseconds. Then execute the following steps in parallel. The task source for these tasks is the dom manipulation task source.
15. While issuedRequests is not empty, perform the following actions depending upon the adjustedTimeout timer and responses from the authenticators:
   If the adjustedTimeout timer expires, For each authenticator in issuedRequests invoke the authenticatorCancel operation on authenticator and remove authenticator from issuedRequests.
   If any authenticator returns a status indicating that the user cancelled the operation, 1. Remove authenticator from issuedRequests.
   2. For each remaining authenticator in issuedRequests invoke the authenticatorCancel operation on authenticator and remove it from issuedRequests.
Relying Party to assess the strength of the credential during later for use, and also contains metadata that can be used by the
This interface represents a newly-created scoped credential. It
interface ScopedCredentialInfo {
interface AuthenticatorResponse {
Authenticators respond to relying party requests by returning an object
This attribute contains a JSON serialization of the client data
A new ArrayBuffer, created using global's %ArrayBuffer%, containing the
bytes of the returned authenticatorData
A new ArrayBuffer, created using global's %ArrayBuffer%, containing the
bytes of the returned signature
A new AuthenticationExtensions object containing the extension identifier -> client extension output entries created by running each extension's client extension processing algorithm to create the client extension outputs, for each client extension in clientDataJSON.clientExtensions.
3. For each remaining authenticator in issuedRequests invoke the authenticatorCancel operation on authenticator and remove it from issuedRequests.
4. Return value and terminate this algorithm.
16. Return a DOMException whose name is "NotAllowedError".
During the above process, the user agent SHOULD show some UI to the user to guide them in the process of selecting and authorizing an authenticator with which to complete the operation.
4.2. Authenticator Responses (interface AuthenticatorResponse)
Authenticators respond to relying party requests by returning an object derived from the AuthenticatorResponse interface:
interface AuthenticatorResponse {
  clientDataJSON, of type ArrayBuffer, readonly
  This attribute contains a JSON serialization of the client data passed to the authenticator by the client in its call to either create() or get().
});
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 Relying Party to assess the strength of the credential during registration.
interface ScopedCredentialInfo {
  [SecureContext]
  readonly attribute ArrayBuffer clientDataJSON;
  readonly attribute ArrayBuffer attestationObject;
};
If any authenticator returns an error status, remove authenticator from issuedRequests.
If any authenticator indicates success,
1. Remove authenticator from issuedRequests.
2. Let value be a new PublicKeyCredential associated with global whose fields are:
[[Identifier]]
A new ArrayBuffer, created using global's %ArrayBuffer%, containing the bytes of the credential ID returned from the successful authenticatorGetAssertion operation, as defined in [Fop-get-assertion]].
response
A new AuthenticatorAssertionResponse object associated with global whose fields are:
clientDataJSON
A new ArrayBuffer, created using global's %ArrayBuffer%, containing the bytes of clientDataJSON
authenticatorData
A new ArrayBuffer, created using global's %ArrayBuffer%, containing the bytes of the returned authenticatorData
signature
A new ArrayBuffer, created using global's %ArrayBuffer%, containing the bytes of the returned signature
clientExtensionResults
A new AuthenticationExtensions object containing the extension identifier -> client extension output entries created by running each extension's client extension processing algorithm to create the client extension outputs, for each client extension in clientDataJSON.clientExtensions.
3. For each remaining authenticator in issuedRequests invoke the authenticatorCancel operation on authenticator and remove it from issuedRequests.
4. Return value and terminate this algorithm.
16. Return a DOMException whose name is "NotAllowedError".
During the above process, the user agent SHOULD show some UI to the user to guide them in the process of selecting and authorizing an authenticator with which to complete the operation.
4.2. Information about Scoped Credential (interface ScopedCredentialInfo)
The clientDataJSON attribute contains the clientDataJSON (see 5.3 Credential Attestation) passed to the authenticator by the client in order to generate this credential. The exact JSON serialization must be preserved as a cryptographic hash (clientDataHash) has been computed over it.

The attestationObject attribute contains an attestation object. The contents of this object are determined by the attestation statement format used by the authenticator. This object is opaque to, and cryptographically protected against tampering by, the client. It contains the credential’s unique identifier, credential public key, and attestation statement. It also contains any additional information that the Relying Party’s server requires to validate the attestation statement, as well as to decode and validate the bindings of both the client and authenticator data. For more details, see 5.3 Credential Attestation.

4.3. User Account Information (dictionary Account)

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

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

The rpDisplayName member contains the friendly name of the Relying Party, such as “Acme Corporation”, “Widgets Inc” or “Awesome Site”.

The id member contains an identifier for the account, specified by the Relying Party. This is not meant to be displayed to the user. It is used by the Relying Party to control the number of credentials an authenticator will never contain more than one credential for a given Relying Party under the same id.

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

The name member contains a detailed name for the account, such as “Acme Corporation”.

4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

The AuthenticatorAttestationResponse interface represents the authenticator’s response to a client’s request for the creation of a new public key credential. It contains information about the new credential that can be used to identify it for later use, and metadata that can be used by the Relying Party to assess the characteristics of the credential during registration.

```
[SecureContext]
interface AuthenticatorAttestationResponse : AuthenticatorResponse {  
  readonly attribute ArrayBuffer attestationObject;  
  clientDataJSON, inherited from AuthenticatorResponse, contains the JSON-serialized client data (see 5.3 Credential Attestation) passed to the authenticator by the client in order to generate this credential. The exact JSON serialization must be preserved, as the hash of the serialized client data has been computed over it.

  attestationObject, of type ArrayBuffer, readonly  
  This attribute contains an attestation object, which is opaque to, and cryptographically protected against tampering by, the client. The attestation object contains both authenticator data and an attestation statement. The former contains the AAGUID, a unique credential ID, and the credential public key. The contents of the attestation statement are determined by the attestation statement format used by the authenticator. It also contains any additional information that the Relying Party’s server requires to validate the attestation statement, as well as to decode and validate the authenticator data along with the JSON-serialized client data. For more details, see 5.3 Credential Attestation as well as Figure 3.

  clientDataJSON
  The clientDataJSON attribute contains the JSON-encoded client data, and is passed to the authenticator when creating a new public key credential.

  clientDataJSON
  This attribute, inherited from AuthenticatorResponse, contains the JSON-serialized client data (see 5.3 Credential Attestation) passed to the authenticator by the client in order to generate this credential. The exact JSON serialization must be preserved, as the hash of the serialized client data has been computed over it.
```

4.2.2. Web Authentication Assertion (Interface AuthenticatorAssertionResponse)

The AuthenticatorAssertionResponse interface represents an authenticator’s response to a client’s request for generation of a new authentication assertion given the Relying Party’s challenge and optional list of credentials it is aware of. This response contains a cryptographic signature proving possession of the credential private key, and optionally evidence of user consent to a specific transaction.

```
[SecureContext]
interface AuthenticatorAssertionResponse : AuthenticatorResponse {  
  readonly attribute ArrayBuffer authenticatorData;  
  readonly attribute ArrayBuffer signature;  
};
```

```
clientDataJSON
```
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.

4.4. Parameters for Credential Generation (dictionary ScopedCredentialParameters)

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

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 signature algorithm with which the newly generated credential will be used, and thus also the type of asymmetric key pair to be generated, e.g., RSA or Elliptic Curve.

4.5. Additional options for Credential Generation (dictionary ScopedCredentialOptions)

dictionary ScopedCredentialOptions {
    unsigned long timeout;
    USVString rpId;
    sequence<ScopedCredentialDescriptor> excludeList = [];
    Attachment attachment;
    AuthenticationExtensions extensions;
};

This dictionary is used to supply additional options when creating a new credential. All these parameters are optional.

* The timeout parameter specifies a time, in milliseconds, that the caller is willing to wait for the call to complete. This is treated

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 signature algorithm with which the newly generated credential will be used, and thus also the type of asymmetric key pair to be generated, e.g., RSA or Elliptic Curve.

4.4. User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity)

dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
    DOMString displayName;
};

This dictionary is used to supply additional parameters about the user account when creating a new credential.

The displayName member contains a friendly name for the user account (e.g., “John P. Smith”).

4.5. Options for Credential Creation (dictionary MakeCredentialOptions)

dictionary MakeCredentialOptions {
    required PublicKeyCredentialEntity rp;
    required PublicKeyCredentialUserEntity user;
    required BufferSource challenge;
    required sequence<PublicKeyCredentialParameters> parameters;
};

This attribute, inherited from AuthenticatorResponse, contains the JSON-serialized client data (see 4.8.1 Client data used in WebAuthn signatures (dictionary CollectedClientData)) passed to the authenticator by the client in order to generate this assertion. The exact JSON serialization must be preserved, as the hash of the serialized client data has been computed over it.

authenticatorData, of type ArrayBuffer, readonly

This attribute contains the authenticator data returned by the authenticator. See 5.1 Authenticator data.

signature, of type ArrayBuffer, readonly

This attribute contains the raw signature returned from the authenticator. See 5.2.2 The authenticatorGetAssertion operation.

4.3. Parameters for Credential Generation (dictionary PublicKeyCredentialParameters)

dictionary PublicKeyCredentialParameters {
    required PublicKeyCredentialType type;
    required AlgorithmIdentifier algorithm;
};

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 signature algorithm with which the newly generated credential will be used, and thus also the type of asymmetric key pair to be generated, e.g., RSA or Elliptic Curve.

4.4. Parameters for Credential Generation (dictionary MakeCredentialOptions)

dictionary MakeCredentialOptions {
    required PublicKeyCredentialEntity rp;
    required PublicKeyCredentialUserEntity user;
    required BufferSource challenge;
    required sequence<PublicKeyCredentialParameters> parameters;
};

This dictionary is used to supply additional options when creating a new credential. All these parameters are optional.

* The timeout parameter specifies a time, in milliseconds, that the caller is willing to wait for the call to complete. This is treated

This attribute contains data about the relying party responsible for the request.
authenticators are eligible to participate in a 4.1.1 Create a new

descriptions, which are used as an additional constraint on which
the same account on a single authenticator. The platform is requested
return an error if the new credential would be created on an
authentication that also contains one of the credentials enumerated
in this parameter.

The extensions parameter contains additional parameters requesting
additional processing by the client and authenticator. For example, the
caller may request that only authenticators with certain
capabilities be used to create the credential, or that particular
information be returned in the attestation object. The caller may
also specify an additional message that they would like the
authenticator to display to the user. Extensions are defined in 8

WebAuthn Extensions.

* The excludeList parameter is intended for use by Relying Parties
that wish to limit the creation of multiple credentials for the
same account on a single authenticator. The platform is requested
to return an error if the new credential would be created on an
authentication that also contains one of the credentials enumerated
in this parameter.

* The extensions parameter contains additional parameters requesting
additional processing by the client and authenticator. For example, the
caller may request that only authenticators with certain
capabilities be used to create the credential, or that particular
information be returned in the attestation object. The caller may
also specify an additional message that they would like the
authenticator to display to the user. Extensions are defined in 8

WebAuthn Extensions.

* The attachment parameter contains authenticator attachment
descriptions, which are used as an additional constraint on which
authenticators are eligible to participate in a 4.1.1 Create a new
credential - makeCredential() method or 4.1.2 Use an existing

WebAuthn Extensions.

* The excludeList parameter is intended for use by Relying Parties
that wish to limit the creation of multiple credentials for the
same account on a single authenticator. The platform is requested
to return an error if the new credential would be created on an
authentication that also contains one of the credentials enumerated
in this parameter.

* The extensions parameter contains additional parameters requesting
additional processing by the client and authenticator. For example, the
caller may request that only authenticators with certain
capabilities be used to create the credential, or that particular
information be returned in the attestation object. Extensions are defined in 8 WebAuthn Extensions.

WebAuthn Extensions.

* The attachment parameter contains authenticator attachment
descriptions, which are used as an additional constraint on which
authenticators are eligible to participate in a 4.1.1 Create a new
credential - makeCredential() method or 4.1.2 Use an existing
roaming authenticators. are defined as having cross-platform attachment, and refer to them as roaming authenticators.

While those that are reachable via cross-platform transport protocols are defined as having cross-platform attachment, and refer to them as platform authenticators. We define authenticators that are part of the client’s platform as having a platform attachment, and refer to them as platform authenticators. While those that are reachable via cross-platform transport protocols are defined as having cross-platform attachment, and refer to them as roaming authenticators.

Clients may communicate with authenticators using a variety of mechanisms. For example, a client may use a platform-specific API to communicate with an authenticator which is physically bound to a platform. On the other hand, a client may use a variety of standardized cross-platform transport protocols such as Bluetooth (see 4.9.5 Credential Transport enumeration (enum ExternalTransport)) to discover and communicate with cross-platform attached authenticators. We define authenticators that are part of the client’s platform as having a platform attachment, and refer to them as platform authenticators.

Clients may communicate with authenticators using a variety of mechanisms. For example, a client may use a platform-specific API to communicate with an authenticator which is physically bound to a platform. On the other hand, a client may use a variety of standardized cross-platform transport protocols such as Bluetooth (see 4.9.5 Credential Transport enumeration (enum ExternalTransport)) to discover and communicate with cross-platform attached authenticators. Therefore, we use Attachment to describe an authenticator’s attachment modality. We define authenticators that are part of the client’s platform as having a platform attachment, and refer to them as platform authenticators. While those that are reachable via cross-platform transport protocols are defined as having cross-platform attachment, and refer to them as roaming authenticators.
platform attachment - the respective authenticator is attached using cross-platform transports. Authenticators of class are non-removable from the platform.
* cross-platform attachment - the respective authenticator is attached using cross-platform transports. Authenticators of this class are removable from, and can "roam" among, client platforms.

This distinction is important because there are use-cases where only platform authenticators are acceptable to a Relying Party, and conversely ones where only roaming authenticators are employed. As a concrete example of the former, a credential on a platform authenticator may be used by Relying Parties to quickly and conveniently reauthenticate the user with a minimum of friction, e.g., the user will not have to dig around in their pocket for their key fob or phone. As a concrete example of the latter, the user is accessing the Relying Party from a given client for the first time, they may be required to use a roaming authenticator which was originally registered with the Relying Party using a different client.

4.6. Web Authentication Assertion (interface AuthenticationAssertion)

```javascript
interface AuthenticationAssertion {
  required attribute ArrayBuffer signature;
  readonly attribute ArrayBuffer authenticatorData;
  readonly attribute ArrayBuffer clientDataJSON;
}
```

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 attribute represents the credential that was used to generate this assertion.

The clientDataJSON attribute contains the parameters sent to the authenticator by the client, in serialized form. See 4.9.1 Client data used in WebAuthn signatures (dictionary ClientData) for the format of this parameter and how it is generated.

The authenticatorData attribute contains the serialized data returned by the authenticator. See 5.1 Authenticator data;

The signature attribute contains the raw signature returned from the authenticator. See 5.2.2 The authenticatorGetAssertion operation.

4.7. Additional options for Assertion Generation (dictionary AssertionOptions)

```javascript
dictionary AssertionOptions {
  unsigned long timeout;
  USVString rpId;
  sequence<ScopedCredentialDescriptor> allowList = [];
  AuthenticationExtensions extensions;
}
```

This dictionary is used to supply additional options when generating an assertion. All these parameters are optional.

- The optional timeout parameter specifies a time, in milliseconds, 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 rpId parameter specifies the rpId claimed by the caller. If it is omitted, it will be assumed to be equal to the current settings object's origin.
- The optional allowList member contains a list of credentials associated with this assertion.

The PublicKeyCredentialRequestOptions dictionary supplies get() with the data it needs to generate an assertion. Its challenge member must be present, while its other members are optional.

```javascript
dictionary PublicKeyCredentialRequestOptions {
  required BufferSource challenge;
  unsigned long timeout;
  USVString rpId;
  sequence<ScopedCredentialDescriptor> allowList = [];
  AuthenticationExtensions extensions;
}
```

This member represents a challenge that the selected authenticator signs, along with other data, when producing an authentication assertion.
4.9.1. Client data used in WebAuthn signatures (dictionary ClientData)

The client data represents the contextual bindings of both the 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.

```idl
dictionary ClientData {
  required DOMString challenge;
  required DOMString origin;
  required AlgorithmIdentifier hashAlg;
  AuthenticationExtensions extensions;
};
```

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

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

The hashAlg member specifies the hash algorithm used to compute the authenticator's attestation.

4.8. Supporting Data Structures

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

4.8.1. Client data used in WebAuthn signatures (dictionary CollectedClientData)

The client data represents the contextual bindings of both the 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.

```idl
dictionary CollectedClientData {
  required DOMString challenge;
  required DOMString origin;
  required AlgorithmIdentifier hashAlg;
  DOMString allowList;  // optional
  AuthenticationExtensions clientExtensions;  // optional
  AuthenticationExtensions authenticatorExtensions;  // optional
};
```

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

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

The hashAlg member is a recognized algorithm name that supports the authentication protocol.

The allowList parameter contains an array of public key credential descriptor objects representing public key credentials acceptable to the caller, in descending order of the caller's preference (the first item in the list is the most preferred credential, and so on down the line).

4.7. Authentication Extensions (typedef AuthenticationExtensions)

typedef record<DOMString, any> AuthenticationExtensions;

This is a dictionary containing zero or more WebAuthn extensions, as defined in 8 WebAuthn Extensions. An AuthenticationExtensions instance can contain either client extensions or authenticator extensions, depending upon context.

4.6. Authentication Assertion Extensions (typedef AuthenticationExtensions)

typedef record<DOMString, any> AuthenticationExtensions;

This optional member 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 might be included as an extension.

4.5. Supporting Data Structures

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

4.4.1. User context used in WebAuthn signatures (dictionary UserContext)

The user context represents the contextual bindings of the user 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.

```idl
dictionary UserContext {
  required DOMString challenge;
  required DOMString origin;
  required AlgorithmIdentifier hashAlg;
  AuthenticationExtensions extensions;
};
```

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

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

The hashAlg member is a recognized algorithm name that supports the authentication protocol.
4.9.3. Unique Identifier for Credential (interface ScopedCredential)

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

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 Authentication Assertion and attestation structures according to the type of the authenticator.

4.9.2. Credential Type enumeration (enum ScopedCredentialType)

enum ScopedCredentialType {
    "ScopedCred",
    "PublicKey"
};

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 Authentication Assertion and attestation structures according to the type of the authenticator.

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

4.8.2. Credential Type enumeration (enum PublicKeyCredentialType)

enum PublicKeyCredentialType {
    "public-key"
};

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 Authentication Assertion and attestation structures according to the type of the authenticator.

Currently one credential type is defined, namely "public-key".

This structure is used by the client to compute the following quantities:

- **clientDataJSON**: This is the UTF-8 encoded JSON serialization [RFC7159] of a ClientData dictionary. Any valid JSON serialization may be used by the client. This specification imposes no canonicalization requirements. Instead, the ScopedCredentialInfo and AuthenticationAssertion structures contain the actual serializations used by the client to generate them.

- **clientDataHash**: This is the hash (computed using hashAlg) of clientDataJSON, as constructed by the client.

The tokenBinding member contains the base64url encoding of the Token Binding ID that this client uses for the Token Binding protocol when communicating with the Relying Party. This can be omitted if no Token Binding has been negotiated between the client and the Relying Party.

The optional extensions member contains additional parameters generated by processing the extensions passed in by the Relying Party. WebAuthn extensions are detailed in Section 8 WebAuthn Extensions.

This structure is used by the client to compute the following quantities:

- **JSON-serialized client data**: This is the UTF-8 encoding of the result of calling the initial value of JSON.stringify on a CollectedClientData dictionary.

This is the hash (computed using hashAlg) of the JSON-serialized client data, as constructed by the client.

The optional clientExtensions and authenticatorExtensions members contain additional parameters generated by processing the extensions passed in by the Relying Party. WebAuthn extensions are detailed in Section 8 WebAuthn Extensions.

This structure is used by the client to compute the following quantities:

- **Hash of the serialized client data**: This is the hash (computed using hashAlg) of the JSON-serialized client data, as constructed by the client.

Currently one credential type is defined, namely "public-key".

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 Authentication Assertion and attestation structures according to the type of the authenticator.

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

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 ScopedCredentialType, 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.

4.9.4. Credential Descriptor (dictionary ScopedCredentialDescriptor)

dictionary ScopedCredentialDescriptor {
  required ScopedCredentialType type;
  required BufferSource id;
  sequence<Transport> transports;
};

This dictionary contains the attributes that are specified by a caller when referring to a credential as an input parameter to the makeCredential() or getAssertion() method. It mirrors the fields of the ScopedCredential object returned by these methods.

The type member contains the type of the credential the caller is referring to.

The id member contains the identifier of the credential that the caller is referring to.

4.9.5. Credential Transport enumeration (enum ExternalTransport)

enum Transport {
  "usb",
  "nfc",
  "ble"
};

Authenticators may communicate with Clients using a variety of transports. This enumeration defines a hint as to how Clients might communicate with a particular Authenticator in order to obtain an assertion for a specific credential. Note that these hints represent the Relying Party’s best belief as to how an Authenticator may be reached. A Relying Party may obtain a list of transports hints from some attestation statement formats or via some out-of-band mechanism; it is outside the scope of this specification to define that mechanism.

* usb - the respective Authenticator may be contacted over USB.
* nfc - the respective Authenticator may be contacted over Near Field Communication (NFC).
* ble - the respective Authenticator may be contacted over Bluetooth Smart (Bluetooth Low Energy / BLE).

4.9.6. 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].

5. 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 authenticators supported by that platform, MUST be indistinguishable from the behavior specified in 4 Web Authentication API.

For authenticators, this model defines the logical operations that they must support, and the data formats that they expose to the client and the Relying Party. However, it does not define the details of how authenticators communicate with the client platform, unless they are required for interoperability with Relying Parties. For instance, requiring Relying Parties to support a particular mechanism does not define protocols for connecting authenticators to clients over transports such as USB or NFC. Similarly, this abstract model does not define specific error codes or methods of returning an error.
them; however, it does define error behavior in terms of the needs of
the client. Therefore, specific error codes are mentioned as a means of
showing which error conditions must be distinguishable (or not) from
each other in order to enable a compliant and secure client
implementation.

In this abstract model, the authenticator provides key management and
cryptographic signatures. It may be embedded in the WebAuthn client, or
hosted in a separate device entirely. The authenticator may choose to
contain a cryptographic module which operates at a higher security
level than the rest of the authenticator. This is particularly
important for authenticators that are embedded in the WebAuthn client,
as those cases this cryptographic module (which may, for example, be
a TPM) could be considered more trustworthy than the rest of the
authenticator.

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 Relying Party, whose identity is
represented by a Relying Party Identifier (RP ID).

Each authenticator has an AAGUID, which is a 128-bit identifier that
indicates the type (e.g. make and model) of the authenticator. The
AAGUID MUST be chosen by the manufacturer to be identical across all
substantially identical authenticators made by that manufacturer, and
different from any other AAGUID. The RP MAY use the AAGUID to infer
the type of authenticator, such as certification level and
strength of key protection, using information from other sources.

The primary function of the authenticator is to provide WebAuthn
signatures, which are bound to various contextual data. These data are
obtained, and added at different levels of the stack as the
request passes from the server to the authenticator. In verifying a
signature, the server checks these bindings against expected values.
These contextual bindings are divided in two: Those added by the RP or
the client, referred to as client data; and those added by the
authenticator, referred to as the authenticator data. The authenticator
signs over the client data, but is otherwise not interested in its
contents. To save bandwidth and processing requirements on the
authenticator, the client hashes the client data and sends only the
result to the authenticator. The authenticator signs over the
combination of this clientDataHash, and its own authenticator data.

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

Authenticators produce cryptographic signatures for two distinct
purposes:

1. An attestation signature is produced when a new credential is
   created, and provides cryptographic proof of certain properties of the
   credential and the authenticator. For instance, an attestation
   signature asserts the type of authenticator (as denoted by its
   AAGUID) and the public key of the credential. The attestation
   signature is signed by an assertion key, which is chosen
   depending on the type of attestation desired. For more details on
   attestation, see 5.3 Credential Attestation.

2. An assertion signature is produced when the
The authenticatorGetAssertion method is invoked. It represents an assertion by the authenticator that the user has consented to a specific transaction, such as logging in, or completing a purchase. Thus, an assertion signature asserts that the authenticator, which possesses a particular credential private key has established, to the best of its ability, that the human who is requesting this transaction is the same human who consented to creating that particular credential. It also provides additional information that might be useful to the caller, such as the means by which user consent was provided, and the prompt that was shown to the user by the authenticator.

The formats of these signatures, as well as the procedures for generating them, are specified below.

5.1. Authenticator data

The authenticator data structure, authenticatorData, encodes contextual bindings made by the authenticator. These bindings are controlled by the authenticator itself, and they rely on the Relying Party's assessment of the security properties of the authenticator. In one extreme case, the authenticator may be embedded in the client, and its bindings may be no more trustworthy than the client. In the other extreme, the authenticator may be a discrete entity with high-security hardware and software, connected to the client over a secure channel. In both cases, the Relying Party receives the authenticator data in the same format, and uses its knowledge of the authenticator to make trust decisions.

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 37 bytes or more, as follows.

Length (in bytes) Description
32 SHA-256 hash of the RP ID associated with the credential.
1 Flags (bit 0 is the least significant bit):
  * Bit 0: Test of User Presence (TUP) result.
  * Bits 1-5: Reserved for future use (RFU).
  * Bit 6: Attestation data included (AT). Indicates whether the authenticator added attestation data.
  * Bit 7: Extension data included (ED). Indicates if the authenticator data has extensions.

4 Signature counter (signCount), 32-bit unsigned big-endian integer. variable (if present) Attestation data (if present). See 5.3.1

20 Authenticator extension outputs as values. See WebAuthn Extensions for details.

The RP ID is originally received from the client when the credential is created and again when an assertion is generated. However, it differs from other client data in some important ways. First, unlike the client data, the RP ID of a credential does not change between operations but instead remains the same for the lifetime of that credential. Secondly, it is validated by the authenticator during the authenticatorGetAssertion operation. Upon verifying that the RP ID associated with the requested credential exactly matches the RP ID supplied by the client.

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.
For attestation signatures, the authenticator MUST set the AT flag and include the attestation data. For authentication signatures, the AT flag MUST NOT be set and the attestation data MUST NOT be included.

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.

![fido-signature-formats-figure1.svg](authenticatorData layout.)

Note that the authenticatorData describes its own length: If the AT and ED flags are not set, it is always 37 bytes long. The attestation data (which is only present if the AT flag is set) describes its own length. If the ED flag is set, then the total length is 37 bytes plus the length of the attestation data, plus the length of the CBOR map that follows.

### 5.2. 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.

#### 5.2.1. The authenticatorMakeCredential operation

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

- The caller’s RP ID, as determined by the user agent and the client.
- The clientDataHash, which is the hash of the serialized ClientData and is provided by the client.
- The Account information provided by the Relying Party.
- A list of PublicKeyCredential objects provided by the Relying Party, with the cryptographic algorithms normalized as per the procedure in Web Cryptography API algorithm-normalization-normalize-an-algorithm.
- A list of ScopedCredential objects provided by the Relying Party with the intention that, if any of these are known to the authenticator, it should not create a new credential.
- Extension data created by the client based on the extensions requested by the authenticator.

When this operation is invoked, the authenticator must perform the following procedure:

- Check if all the supplied parameters are syntactically well-formed and of the correct length. If not, return an error code equivalent to UnknownError and terminate the operation.
- Check if at least one of the specified combinations of ScopedCredentialType and cryptographic parameters is supported. If not, return an error code equivalent to NotSupportedError and terminate the operation.
- Check if a credential matching any of the supplied ScopedCredential identifiers is present on this authenticator. If so, return an error code equivalent to NotAllowedError and terminate the operation.
- Prompt the user for consent to create 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. If the

---

For attestation signatures, the authenticator MUST set the AT flag and include the attestation data. For authentication signatures, the AT flag MUST NOT be set and the attestation data MUST NOT be included.

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.

![fido-signature-formats-figure1.html](authenticator data layout.)

Note that the authenticator data describes its own length: If the AT and ED flags are not set, it is always 37 bytes long. The attestation data (which is only present if the AT flag is set) describes its own length. If the ED flag is set, then the total length is 37 bytes plus the length of the attestation data, plus the length of the CBOR map that follows.

### 5.2.2. The authenticatorGetCredentials operation

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

- The caller’s RP ID, as determined by the user agent and the client.
- The hash of the serialized ClientData, provided by the client.
- The relying party’s PublicKeyCredentialEntity.
- The user account’s PublicKeyCredentialEntity.
- A list of PublicKeyCredential objects provided by the Relying Party with the intention that, if any of these are known to the authenticator, it should not create a new credential.
- Extension data created by the client based on the extensions requested by the authenticator.

When this operation is invoked, the authenticator must perform the following procedure:

- Check if all the supplied parameters are syntactically well-formed and of the correct length. If not, return an error code equivalent to UnknownError and terminate the operation.
- Check if at least one of the specified combinations of PublicKeyCredentialType and cryptographic parameters is supported. If not, return an error code equivalent to NotSupportedError and terminate the operation.
- Check if a credential matching any of the supplied PublicKeyCredential identifiers is present on this authenticator. If so, return an error code equivalent to NotAllowedError and terminate the operation.
- Prompt the user for consent to create 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. If the
user declines consent, return an error code equivalent to NotAllowedError and terminate the operation.
* Once user consent has been obtained, generate a new credential object:
  + Generate a set of cryptographic keys using the most preferred combination of PublicKeyCredentialType and cryptographic parameters supported by this authenticator.
  + Generate an identifier for this credential, such that this identifier is globally unique with high probability across all credentials with the same type across all authenticators.
  + Associate the credential with the specified RP ID and the user’s account identifier id that are stored locally in the authenticator.
  * If any error occurred while creating the new credential object, return an error code equivalent to UnknownError and terminate the operation.
  + If the previous step resulted in an empty list, return an error code equivalent to NotAllowedError and terminate the operation.
  * Process all the supported extensions requested by the client, and generate an authenticatorData structure with attestation data as specified in 5.1 Authenticator data. Use this authenticatorData and the clientDataHash received from the client to create an attestation object for the new credential using the procedure specified in 5.3 Credential Attestation.
On successful completion of this operation, the authenticator returns the attestation object to the client.

5.2.2. 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 caller’s RP ID, as determined by the user agent and the client.
* A list of credentials acceptable to the Relying Party (possibly filtered by the client).
* Extension data created by the client based on the extensions requested by the Relying Party.

When this method is invoked, the authenticator must perform the following procedure:
* Check if all the supplied parameters are syntactically well-formed and of the correct length. If not, return an error code equivalent to UnknownError and terminate the operation.
* If a list of credentials was supplied by the client, filter it by removing those credentials that are not present on this authenticator. If no list was supplied, create a list with all credentials stored for the caller’s RP ID (as determined by an exact match of the RP ID).
* If the previous step resulted in an empty list, return an error code equivalent to NotAllowedError and terminate the operation.
* Otherwise, the user to select a credential from among the above list.
* Obtain user consent for using this 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.
* Process all the supported extensions requested by the client, and generate an authenticatorData structure without attestation data as specified in 5.1 Authenticator data. Concatenate this authenticatorData with the clientDataHash received from the client to generate an assertion signature using the private key of the selected credential as shown below. A simple, undeclared 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.

* If any error occurred while generating the assertion signature, return an error code equivalent to UnknownError and terminate the operation.
* Once user consent has been obtained, generate a new credential object:
  + Generate a set of cryptographic keys using the most preferred combination of PublicKeyCredentialType and cryptographic parameters supported by this authenticator.
  + Generate an identifier for this credential, such that this identifier is globally unique with high probability across all credentials with the same type across all authenticators.
  + Associate the credential with the specified RP ID and the user’s account identifier id that are stored locally in the authenticator.
  * If any error occurred while creating the new credential object, return an error code equivalent to UnknownError and terminate the operation.
  + If the previous step resulted in an empty list, return an error code equivalent to NotAllowedError and terminate the operation.
  * Process all the supported extensions requested by the client, and generate an authenticatorData with attestation data as specified in 5.1 Authenticator data. Use this authenticatorData and the hash of the serialized client data to create an attestation object for the new credential using the procedure specified in 5.3 Credential Attestation.
On successful completion of this operation, the authenticator returns the attestation object to the client.

5.2.2. 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 caller’s RP ID, as determined by the user agent and the client.
* A list of credentials acceptable to the Relying Party (possibly filtered by the client).
* Extension data created by the client based on the extensions requested by the Relying Party.

When this method is invoked, the authenticator must perform the following procedure:
* Check if all the supplied parameters are syntactically well-formed and of the correct length. If not, return an error code equivalent to UnknownError and terminate the operation.
* If a list of credentials was supplied by the client, filter it by removing those credentials that are not present on this authenticator. If no list was supplied, create a list with all credentials stored for the caller’s RP ID (as determined by an exact match of the RP ID).
* If the previous step resulted in an empty list, return an error code equivalent to NotAllowedError and terminate the operation.
* Otherwise, the user to select a credential from among the above list.
* Obtain user consent for using this 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.
* Process all the supported extensions requested by the client, and generate an authenticatorData structure without attestation data as specified in 5.1 Authenticator data. Concatenate this authenticatorData with the clientDataHash received from the client to generate an assertion signature using the private key of the selected credential as shown below. A simple, undeclared concatenation is safe to use here because the authenticatorData describes its own length. The hash of the serialized client data (which potentially has a variable length) is always the last element.

* If any error occurred while generating the assertion signature, return an error code equivalent to UnknownError and terminate the operation.
On successful completion, the authenticator returns to the user agent:

- The identifier of the credential used to generate the signature.
- The authenticatorData used to generate the signature.
- The assertion signature.

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

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

5.2.3. 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.

5.3. Credential Attestation

Authenticators must 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 Relying Party. Typically, this information contains a signature by an attestation private key over the attested credential public key and a challenge, as well as a certificate or similar information providing provenance information for the attestation public key, enabling a trust decision to be made. However, if an attestation key pair is not available, then the authenticator MUST perform self attestation of the credential public key with the corresponding credential private key. All this information is returned by the authenticator any time a new credential is generated, in the form of an attestation object. The relationship of authenticator data and the attestation data, attestation object, and attestation statement data structures is illustrated in the figure below.

An important component of the attestation object is the credential attestation statement. This is a specific type of signed data object, containing statements about a credential itself and the authenticator that created it. It contains an attestation signature created using the key of the attesting authority (except for the case of self attestation, when it is created using the private key associated with the credential). To correctly interpret an attestation statement, a Relying Party needs to understand two aspects of the attestation:

1. The attestation statement format is the manner in which the signature is represented and the various contextual bindings are incorporated into the attestation statement by the authenticator. In other words, this defines the syntax of the statement. Various existing devices and platforms (such as Windows and Android OS) have previously defined attestation statement formats. This specification supports a variety of such formats in an extensible way, as defined in 5.3.2 Attestation Statement Formats.
2. The attestation type defines the semantics of the attestation statement and its underlying trust model. It defines how a Relying Party can verify the authenticity of the attestation data contained in the attestation statement and whether or not it is valid.
As described above, an attestation statement format is a data format that supports a number of attestation types, as described in 5.3.3 Attestation Types.

In general, there is no simple mapping between attestation statement formats and attestation types. For example, the "packed" attestation statement format defined in 7.2 Packed Attestation Statement Format can be used in conjunction with all attestation types, while other formats and types have more limited applicability.

The privacy, security and operational characteristics of attestation types depend on:
* The attestation type, which determines the trust model,
* The attestation statement format, which may constrain the strength of the attestation by limiting what can be expressed in an attestation statement,
* The characteristics of the individual authenticator, such as its construction, whether part or all of it runs in a secure operating environment, and so on.

It is expected that most authenticators will support a small number of attestation types and attestation statement formats, while Relying Parties will decide what attestation types are acceptable to them by policy. Relying Parties will also need to understand the characteristics of the authenticators that they trust, based on information they have about these authenticators. For example, the FIDO Metadata Service [FIDOMetadataService] provides one way to access such information.

### 5.3.1. Attestation data

Attestation data is added to the authenticatorData when generating an attestation object for a given credential. It has the following format:

#### Length (in bytes) Description
- 16 The AAGUID of the authenticator.
- 2 Byte length L of Credential ID
- L Credential ID variable Credential public key encoded in CBOR format. This is a CBOR map defined by the following CDDL rules:

```
pubKey = SpubKeyFmt

pubKeyFmt .within $pubKeyFmt

pubKeyFmt /= rsaPubKey

rsaPubKey = { alg: rsaAlgName, n: biguint, e: uint }

rsaAlgName = "RS256" / "RS384" / "RS512" / "PS256" / "PS384" / "PS51"

2"
```

Thus, each public key type is a CBOR map starting with an entry named alg, which contains a text string that specifies the signature algorithm associated with the credential private key, using values defined in [RFC7518] section 3.1. The semantics and naming of the other fields (though not their encoding) follows the definitions in [RFC7518] section 6. Specifically, for ECC keys, the semantics of the x and y fields are defined in [RFC7518] sections 6.2.1.2 and 6.2.1.3, while for RSA keys, the semantics of the n and e fields are defined in [RFC7518] sections 6.3.1.1 and 6.3.1.2.

### 5.3.2. Attestation Statement Formats

As described above, an attestation statement format is a data format which represents a cryptographic signature by an authenticator over a party establishes trust in a particular attestation statement, after verifying that it is cryptographically valid. This specification supports a number of attestation types, as described in 5.3.3 Attestation Types.

In general, there is no simple mapping between attestation statement formats and attestation types. For example, the "packed" attestation statement format defined in 7.2 Packed Attestation Statement Format can be used in conjunction with all attestation types, while other formats and types have more limited applicability.

The privacy, security and operational characteristics of attestation have about these authenticators. For example, the FIDO Metadata Service [FIDOMetadataService] provides one way to access such information.

#### Length (in bytes) Description
- 16 The AAGUID of the authenticator.
- 2 Byte length L of Credential ID
- L Credential ID variable Credential public key encoded in CBOR format. This is a CBOR map defined by the following CDDL rules:

```
pubKey = SpubKeyFmt

pubKeyFmt .within $pubKeyFmt

pubKeyFmt /= rsaPubKey

rsaPubKey = { alg: rsaAlgName, n: biguint, e: uint }

rsaAlgName = "RS256" / "RS384" / "RS512" / "PS256" / "PS384" / "PS51"

2"
```

Thus, each public key type is a CBOR map starting with an entry named alg, which contains a text string that specifies the signature algorithm associated with the credential private key, using values defined in [RFC7518] section 3.1. The semantics and naming of the other fields (though not their encoding) follows the definitions in [RFC7518] section 6. Specifically, for ECC keys, the semantics of the x and y fields are defined in [RFC7518] sections 6.2.1.2 and 6.2.1.3, while for RSA keys, the semantics of the n and e fields are defined in [RFC7518] sections 6.3.1.1 and 6.3.1.2.

### 5.3.2. Attestation Statement Formats

As described above, an attestation statement format is a data format which represents a cryptographic signature by an authenticator over a
set of contextual bindings. Each attestation statement format is defined by the following attributes:

- Its attestation statement format identifier.
- The set of attestation types supported by the format.
- The syntax of an attestation statement produced in this format, defined using CDDL for the extension point SatStmtFormat defined in 5.3.4 Generating an Attestation Object.
- The procedure for computing an attestation statement in this format given the credential to be attested, the authenticatorData for the attestation, and a clientDataHash.
- The procedure for verifying an attestation statement, which takes as inputs the authenticatorData claimed to have been used for the attestation and the clientDataHash of the client’s contextual bindings, and returns either:
  + An error indicating that the attestation is invalid, or
  + The attestation type, and the trust path of the attestation.

This trust path is either empty (in case of self-attestation), a DAA root key (in the case of Direct Anonymous Attestation), or a set of X.509 certificates.

The initial list of supported attestation statement formats is in 7 Defined Attestation Statement Formats.

5.3.3. Attestation Types

WebAuthn supports multiple attestation types:

Basic Attestation

In the case of basic attestation (UAFProtocol), the authenticator’s attestation key pair is specific to an authenticator model. Thus, authenticators of the same model often share the same attestation key pair. See 5.3.5.1 Privacy for further information.

Self Attestation

In the case of self-attestation, also known as surrogate basic attestation (UAFProtocol), the Authenticator doesn’t have any specific attestation key, instead it uses the authentication key itself to create the attestation signature. Authenticators without meaningful protection measures for an attestation private key typically use this attestation type.

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 CAs(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.
5.3.4. Generating an Attestation Object

This section specifies the algorithm for generating an attestation object for any attestation statement format.

In order to construct an attestation object for a given credential using a particular attestation statement format, the authenticator MUST first generate an authenticatorData structure.

The authenticator MUST then run the signing procedure for the desired attestation statement format with this authenticatorData and the client-supplied clientDataHash as input, and use this to construct an attestation statement in that attestation statement format.

Finally, the authenticator MUST construct the attestation object as a CBOR map with the following syntax:

\[
\text{attObj} = \{ \\
\quad \text{authData: bytes,} \\
\quad \$attStmtType \\
\} \\
\text{attStmtTemplate} = ( \\
\quad \text{fmt: text,} \\
\quad \text{attStmt: bytes} \\
\) \\
\]

Every attestation statement format must have the above fields.

The semantics of the fields in the attestation object are as follows:

- **fmt**: The attestation statement format identifier associated with the attestation statement. Each attestation statement format defines its identifier.
- **authData**: The authenticator data used to generate the attestation statement.
- **attStmt**: The attestation statement constructed above. The syntax of this is defined by the attestation statement format used.

5.3.5. Security Considerations

5.3.5.1. 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 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 approach). 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 [FIDOEdcdaaAlgorithm]). Using this scheme, the authenticator...
these structures.

the operations that the Relying Party must perform upon receipt of these structures. This section describes the operations that the Relying Party must perform upon receipt of these structures.

5.3.5.2. 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 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.) If the WebAuthn Authenticator manufacturer does not have the capability, then it may not be possible for Relying Parties to trust any further attestation statements from the affected WebAuthn Authenticators.

If attestation certificate validation fails due to a revoked intermediate attestation CA certificate, and the Relying Party's policy requires rejecting the registration/authentication request in these situations, then it is recommended that the Relying Party also un-register (or mark with a trust level equivalent to “self attestation”) scoped credentials that were registered after the CA compromise date using an attestation certificate chaining up to the same intermediate CA. It is thus recommended that Relying Parties remember intermediate attestation CA certificates during Authenticator registration in order to un-register related 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 Relying Party should verify whether an authenticator belongs to the RogueList when performing DAA-Verify. For example, the FIDO Metadata Service [FIDOMetadataService] provides one way to access such information.

5.3.5.3. Attestation Certificate Hierarchy

5-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 isolate attestation keys for their devices can issue new attestation certificates for these keys from a new intermediate CA or from a new root CA. If the root CA changes, the Relying Parties must update their trusted root certificates accordingly.

A 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.) If the WebAuthn Authenticator manufacturer does not have the capability, then it may not be possible for Relying Parties to trust any further attestation statements from the affected WebAuthn Authenticators.

If attestation certificate validation fails due to a revoked intermediate attestation CA certificate, and the Relying Party's policy requires rejecting the registration/authentication request in these situations, then it is recommended that the Relying Party also un-register (or mark with a trust level equivalent to “self attestation”) public key credentials that were registered after the CA compromise date using an attestation certificate chaining up to the same intermediate CA. It is thus recommended that Relying Parties remember intermediate attestation CA certificates during Authenticator registration in order to un-register related public key credentials if the registration was performed after revocation of such certificates.

If an ECDAA attestation key has been compromised, it can be added to the RogueList (i.e., the list of revoked authenticators) maintained by the related ECDAA-issuer. The Relying Party should verify whether an authenticator belongs to the RogueList when performing ECDAA-Verify (see section 5.3.5.2.1 [FIDOEd25519Algorithm]). For example, the FIDO Metadata Service [FIDOEd25519Algorithm] provides one way to access such information.

5.3.5.3. 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 isolate attestation keys for their devices can issue new attestation certificates for these keys from a new intermediate CA or from a new root CA. If the root CA changes, the Relying Parties must update their trusted root certificates accordingly.

A 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 for their devices can issue new attestation certificates for these keys from a new intermediate CA or from a new root CA. If the root CA changes, the Relying Parties must update their trusted root certificates accordingly.

If attestation certificate validation fails due to a revoked intermediate attestation CA certificate, and the Relying Party's policy requires rejecting the registration/authentication request in these situations, then it is recommended that the Relying Party also un-register (or mark with a trust level equivalent to “self attestation”) public key credentials that were registered after the CA compromise date using an attestation certificate chaining up to the same intermediate CA. It is thus recommended that Relying Parties remember intermediate attestation CA certificates during Authenticator registration in order to un-register related public key credentials if the registration was performed after revocation of such certificates.

If an ECDAA attestation key has been compromised, it can be added to the RogueList (i.e., the list of revoked authenticators) maintained by the related ECDAA-issuer. The Relying Party should verify whether an authenticator belongs to the RogueList when performing ECDAA-Verify (see section 5.3.5.2.1 [FIDOEd25519Algorithm]). For example, the FIDO Metadata Service [FIDOEd25519Algorithm] provides one way to access such information.

5.3.5.3. 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 isolate attestation keys for their devices can issue new attestation certificates for these keys from a new intermediate CA or from a new root CA. If the root CA changes, the Relying Parties must update their trusted root certificates accordingly.

A 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 for their devices can issue new attestation certificates for these keys from a new intermediate CA or from a new root CA. If the root CA changes, the Relying Parties must update their trusted root certificates accordingly.

If attestation certificate validation fails due to a revoked intermediate attestation CA certificate, and the Relying Party's policy requires rejecting the registration/authentication request in these situations, then it is recommended that the Relying Party also un-register (or mark with a trust level equivalent to “self attestation”) public key credentials that were registered after the CA compromise date using an attestation certificate chaining up to the same intermediate CA. It is thus recommended that Relying Parties remember intermediate attestation CA certificates during Authenticator registration in order to un-register related public key credentials if the registration was performed after revocation of such certificates.

If an ECDAA attestation key has been compromised, it can be added to the RogueList (i.e., the list of revoked authenticators) maintained by the related ECDAA-issuer. The Relying Party should verify whether an authenticator belongs to the RogueList when performing ECDAA-Verify (see section 5.3.5.2.1 [FIDOEd25519Algorithm]). For example, the FIDO Metadata Service [FIDOEd25519Algorithm] provides one way to access such information.
6.1. Registering a new credential

When registering a new credential, represented by a

1. Perform JSON deserialization on the clientDataJSON field of the

2. Verify that the challenge in C matches the challenge that was sent to the Authenticator in the makeCredential() call.
3. Verify that the origin in C matches the Relying Party's origin.
4. Verify that the tokenBinding in C matches the tokenBindingId for the TLS connection over which the attestation was obtained.
5. Verify that the extensions in C is a proper subset of the extensions requested by the RP.
6. Compute the hash of clientDataJSON using the hashAlg algorithm found in C.
7. Perform CBOR decoding on the attestationObject field of the

8. Verify that the RP ID hash in authData is indeed the SHA-256 hash of the RP ID expected by the RP.
9. Determine the attestation statement format by performing an ASCII case-insensitive match on fmt against the set of WebAuthn Attestation Statement Format Identifiers given in the IANA Registry of the same name [WebAuthn-Registrries].
10. Verify that attStmt is a correct, validly-signed attestation statement, using the attestation statement format fmt's verification procedure given authenticator data authData and the clientDataHash computed in step 6.
11. If validation is successful, obtain a list of acceptable trust anchors (attestation root certificates or DAA root keys) for that attestation type and attestation statement format fmt, from a trusted source or from policy. For example, the FIDO Metadata Service (FIDOMetadataService) provides one way to obtain such information, using the AAGUID in the attestation data contained in authData.
12. Assess the attestation trustworthiness using the outputs of the verification procedure in step 10, as follows:
   a. If self-attestation was used, check if self-attestation is acceptable under Relying Party policy.
   b. If DAA was used, verify that the DAA key used is in the set of acceptable trust anchors obtained in step 11. Otherwise, use the X.509 certificates returned by the verification procedure to verify that the root certificate that the attestation public key correctly chains up to an acceptable root certificate.
13. If the attestation statement attStmt verified successfully and is found to be trustworthy, then register the new credential with the account that was denoted in the accountInfo passed to makeCredential(), by associating it with the credentialId and clientDataHash contained in authData's attestation data, as appropriate for the Relying Party's systems.
14. If the attestation statement attStmt successfully verified but is not trustworthy per step 12 above, the Relying Party SHOULD fail the registration ceremony.

NOTE: However, if permitted by policy, the Relying Party MAY register the credential Id and credential public key but treat the credential as one with self-attestation (see 5.3 Attestation Type). If doing so, the Relying Party is asserting that no cryptographic proof that the Scopred Credential has been generated by a particular Authenticator model. See [FIDOSecRef] and [UAAProtocol] for a more detailed discussion.
15. If verification of the attestation statement failed, the Relying Party MUST fail the registration ceremony.
Verification of attestation objects requires that the Relying Party has a trusted method of determining acceptable trust anchors in step 11 above. Also, if certificates are being used, the Relying Party must have access to certificate status information for the intermediate CA certificates. The Relying Party must also be able to build the attestation certificate chain if the client did not provide this chain in the attestation information.

To avoid ambiguity during authentication, the Relying Party SHOULD check that each credential is registered to no more than one user. If registration is requested for a credential that is already registered to a different user, the Relying Party SHOULD fail this ceremony, or it MAY decide to accept the registration, e.g. while deleting the older registration.

### 7. Defined Attestation Statement Formats

WebAuthn supports pluggable attestation statement formats. This section defines an initial set of such formats.

#### 7.1. Attestation Statement Format Identifiers

Attestation statement formats are identified by a string, called a attestation statement format identifier, chosen by the author of the attestation statement format. All registered attestation statement format identifiers are unique amongst themselves as a matter of course.

Unregistered attestation statement format identifiers SHOULD use reverse domain-name naming, using a domain name registered by the [WebAuthn-Registries] "Registries for Web Authentication (WebAuthn)". All registered attestation statement format identifiers are unique amongst themselves as a matter of course.

Unregistered attestation statement format identifiers SHOULD use lowercase reverse domain-name naming, using a domain name registered by
Attestation statements that may exist in multiple versions SHOULD include a version in their identifier. In effect, different versions are thus treated as different formats, e.g., packed2 as a new version of the packed attestation statement format.

The following sections present a set of currently-defined and registered attestation statement formats and their identifiers. See the WebAuthn Attestation Statement Format Identifier Registry defined in [WebAuthn-Registries] for an up-to-date list of registered attestation statement formats.

7.2. Packed Attestation Statement Format

This is a WebAuthn optimized attestation statement format. It uses a very compact but still extensible encoding method. It is implementable by authenticators with limited resources (e.g., secure elements).

Attestation statement format identifier

packed

Attestation types supported

All

Syntax

The syntax of a Packed Attestation statement is defined by the following CDDL:

```
$attsStmtType ::= {
  fmt: "packed",
  attStmt: packedStmtFormat

packedStmtFormat ::= {
  alg: rsaAlgName / eccAlgName,
  sig: bytes,
  x5c: [ attestnCert: bytes, * (caCert: bytes) ]
}
```

The semantics of the fields are as follows:

```
alg
A text string containing the name of the algorithm used to generate the attestation signature. The types rsaAlgName and eccAlgName are as defined in 5.3.1 Attestation data. "ED256" and "ED512" refer to algorithms defined in [FIDOEd256 Algorithm].

sig
A byte string containing the attestation signature.

x5c
A byte string containing the attestation signature.
```

The elements of this array contain the attestation certificate and its certificate chain, each encoded in X.509 format. The attestation certificate must be the first element in the array.

daKey

The DAA root key. The syntax for eccPubKey is defined in 5.3.1 Attestation data.

Signning procedure

The signing procedure for this attestation statement format is similar to the procedure for generating assertion signatures.

If Basic or Privacy CA attestation is in use, the authenticator produces the sig by concatenating the given authenticatorData and clientDataHash, and signing the result using an attestation private key selected through an authenticator-specific mechanism. It sets x5c to the certificate chain of the attestation public key and alg to the algorithm of the attestation private key.

If DAA is in use, the authenticator produces sig by concatenating the given authenticatorData and clientDataHash, and signing the result using the DAA root key and daaKey to the DAA root key.

If self attestation is in use, the authenticator produces sig by concatenating the given authenticatorData and clientDataHash, and signing the result using the credential private key. It sets alg to the algorithm of the credential private key, and omits the other fields.

Verification procedure

Verify that the given attestation statement is valid CBOR conforming to the syntax defined above.

If x5c is present, this indicates that the attestation type is not DAA. In this case:

+ Verify that sig is a valid signature over the concatenation of the given authenticatorData and clientDataHash, and the concatenation of authenticatorData public key in x5c with the algorithm specified in alg.
+ Verify that x5c meets the requirements in 7.2.1 Packed attestation statement certificate requirements.
+ If x5c contains an extension with OId 1 3 6 1 4 1 54724 1 1 4 (id-fido-gen-ce-aaguid) verify that the value of this extension matches the AAGUID in the claimed authenticatorData.
+ If successful, return attestation type Basic and trust path x5c.

If daaKey is present, then the attestation type is DAA. In this case:

...
The syntax of a TPM Attestation statement is as follows:

```
Attestation types supported

tpm
```

Attestation statement format identifier that use a Trusted Platform Module as their cryptographic engine.

This attestation statement format is generally used by authenticators, for example, the FIDO Metadata Service [FIDOMetadataService].

### 7.3. TPM Attestation Statement Format

The attestation certificate MUST have the following fields/extensions:

* Subject field MUST be set to:
  * Version must be set to 3.
  * Subject field MUST be set to:
    * Country where the Authenticator vendor is incorporated
    * Legal name of the Authenticator vendor
    * 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 many attestation certificates is available through authenticator metadata services. See, for example, the FIDO Metadata Service [FIDOMetadataService].

### 7.2.1. Packed attestation statement certificate requirements

The attestation certificate MUST have the following fields/extensions:

* Subject field MUST be set to:
  * Version must be set to 3.
  * Subject field MUST be set to:
    * Country where the Authenticator vendor is incorporated
    * Legal name of the Authenticator vendor
    * 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 many attestation certificates is available through authenticator metadata services. See, for example, the FIDO Metadata Service [FIDOMetadataService].

### 7.3. TPM Attestation Statement Format

This attestation statement format is generally used by authenticators that use a Trusted Platform Module as their cryptographic engine.

### Attestation statement format identifier

```
tpm
```

### Attestation types supported

Privacy CA, DAA

### Syntax

The syntax of a TPM Attestation statement is as follows:

```
[...]
```

If neither x5c nor daaKey is present, self attestation is in use.

+ Validate that alg matches the algorithm of the credential private key in the claimed authenticatorData.
+ Verify that sig is a valid signature over the concatenation of authenticatorData and clientDataHash using the credential public key with alg.
+ If successful, return attestation type Self and empty trust path.

If neither x5c nor daaKeyId is present, self attestation is in use.

+ Validate that alg matches the algorithm of the credential private key in authenticatorData.
+ Verify that sig is a valid signature over the concatenation of authenticatorData and clientDataHash using the credential public key with alg.
+ If successful, return attestation type Self and empty trust path.

### 7.2.1. Packed attestation statement certificate requirements

The attestation certificate MUST have the following fields/extensions:

* Version must be set to 3.
* Subject field MUST be set to:
  * Country where the Authenticator vendor is incorporated
  * Legal name of the Authenticator vendor
  * 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 many attestation certificates is available through authenticator metadata services. See, for example, the FIDO Metadata Service [FIDOMetadataService].

### 7.3. TPM Attestation Statement Format

This attestation statement format is generally used by authenticators that use a Trusted Platform Module as their cryptographic engine.

### Attestation statement format identifier

```
tpm
```

### Attestation types supported

Privacy CA, ECDAA

### Syntax

The syntax of a TPM Attestation statement is as follows:
Concatenate the given authenticatorData and clientDataHash to form attToBeSigned.
Generate a signature using the procedure specified in [TPMv2-Parts] section 18.2 using the attestation public key and setting the qualifying-data parameter to attToBeSigned.

Set the pubArea field to the public area of the credential public key, the certInfo field to the output parameter of the same name, and the sig field to the signature obtained from the above procedure.

Verification procedure

Verify that the given attestation statement is valid CBOR conforming to the syntax defined above.

Verify that the public key specified by the parameters and unique fields of pubArea are identical to the public key contained in the attestation data inside the claimed authenticatorData.

Concatenate the given authenticatorData and clientDataHash to form attToBeSigned.

Validate that certInfo is valid:
+ Verify that magic is set to TPM GENERATED VALUE.
+ Verify that type is set to TPM ST ATTEST CERTIFY.
+ Verify that extraData is set to attToBeSigned.
+ Verify that attested contains a TPMS CERTIFY INFO structure, whose name field contains a valid Name for pubArea, as computed using the algorithm in the TPMv2-Part1 section 16.

If x5c is present, this indicates that the attestation type is not DAA. In this case:
+ Verify the sig is a valid signature over certInfo using the attestation public key in x5c with the algorithm specified in alg.
+ Verify that x5c meets the requirements in 7.3.1 TPM attestation statement certificate requirements.
+ If x5c 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 the AAGUID in the claimed authenticatorData.
+ If successful, return attestation type Privacy CA and trust path x5c.

If daaKey is present, then the attestation type is DAA.
+ Perform DAA-Verify on sig to verify that it is a valid signature over certInfo (see [FIDOEdcDaalAlgorithm]).
+ If successful, return attestation type DAA and trust path x5c.

7.3.1. 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-EX-Profile] section 3.2.9.
  * 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

7.3.1.1. 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-EX-Profile] section 3.2.9.
  * 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 many attestation certificates is available through metadata services. See, for example, the FIDO Metadata Service [FIDO_Metadata_Service].

7.4. Android Key Attestation Statement Format

When the Authenticator in question is a platform-provided Authenticator on the Android "N" or later platform, the attestation statement is based on the Android key attestation. In these cases, the attestation statement is produced by a component running in a secure operating environment, but the authenticatorData is produced outside this environment. The Relying Party is expected to check that the contents of authenticatorData are consistent with the fields of the attestation certificate's extension data.

Attestation statement format identifier

android-key

Attestation types supported

Basic

Syntax

An Android key attestation statement consists simply of the Android attestation statement, which is a series of DER encoded X.509 certificates. See the Android developer documentation. Its syntax is defined as follows:

$$attStmtType //= (                        
 | format: "android-key", 
 | attStmt: androidStmtFormat 
 )$$

androidStmtFormat = bytes

Signing procedure

Concatenate the given authenticatorData and clientDataHash to form attToBeSigned.

Request a Android Key Attestation by calling

"keyStore.getCertificateChain(myKeyUUID)" providing attToBeSigned as the challenge value (e.g., by using setAttestationChallenge), and set the attestation statement to the returned value.

Verification procedure

Verification is performed as follows:

+ Let authenticatorData denote the authenticator data for the attestation, and let clientDataHash denote the hash of the serialized client data.

+ Let authenticatorData denote the authenticator data claimed to have been used for the attestation, and let clientDataHash denote the hash of the serialized client data.

+ Let the authenticatorData in the attestation certificate extension data be signed with the trust anchor of the platform, and let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.

+ Let the attestation challenge be the concatenation of the claimed authenticatorData and clientDataHash.

+ Let the attestation statement reference the authenticator data.
Verification is performed as follows:

**Verification procedure**

1. Concatenate the given authenticatorData and clientDataHash to form attToBeSigned.

2. Request a SafetyNet attestation, providing attToBeSigned as the nonce value. Set response to the result, and ver to the version of Google Play Services running in the authenticator.

3. **Verification procedure**

   Verification is performed as follows:

   - The value in the AuthorizationList.origin field is equal to KM_PURPOSE_SIGN.
   - The value in the AuthorizationList.purpose field is equal to KM_TAG_GENERATED.
   - If successful, return attestation type Basic with the trust path set to the entire attestation statement.

---

### 7.5. Android SafetyNet Attestation Statement Format

When the Authenticator in question is a platform-provided Authenticator on certain Android platforms, the attestation statement is based on the SafetyNet API. In this case the authenticator data is completely controlled by the caller of the SafetyNet API (typically an application running on the Android platform) and the attestation statement only provides some statements about the health of the platform and the identity of the calling application.

**Attestation statement format identifier**

- **android-safetynet**

**Attestation types supported**

- Basic

**Syntax**

The syntax of an Android Attestation statement is defined as follows:

```
$$attsStmtType /=
  fmt: "android-safetynet",
  attStmt: safetynetStmtFormat
)

safetynetStmtFormat = {
  ver: text,
  response: bytes
}
```

The semantics of the above fields are as follows:

- **ver**
  - The version number of Google Play Services responsible for providing the SafetyNet API.

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

**Signing procedure**

Let authenticatorData denote the authenticator data for the attestation, and let clientDataHash denote the hash of the serialized client data.

```
Concatenate authenticatorData and clientDataHash to form attToBeSigned.
```

- Request a SafetyNet attestation, providing attToBeSigned as the nonce value. Set response to the result, and ver to the version of Google Play Services running in the authenticator.

- **Verification procedure**

  Verification is performed as follows:

  - The value in the AuthorizationList.origin field is equal to KM_PURPOSE_SIGN.
  - The value in the AuthorizationList.purpose field is equal to KM_TAG_GENERATED.
  - If successful, return attestation type Basic with the trust path set to the entire attestation statement.

---
+ Verify that the given attestation statement is valid CBOR conforming to the syntax defined above.
+ Verify that response is a valid SafetyNet response of version ver.
+ Verify that the nonce in the response is identical to the concatenation of the clientDataHash.
+ Verify that the attestation certificate is issued to the hostname "attest.android.com" (see SafetyNet online documentation).
+ Verify that the ctsProfileMatch attribute in the payload of response is true.
+ If successful, return attestation type Basic with the trust path set to the above attestation certificate.

7.6. FIDO U2F Attestation Statement Format

This attestation statement format is used with FIDO U2F authenticators using the formats defined in [FIDO-U2F-Message-Formats].

Attestation statement format identifier

* fido-u2f

Attestation types supported

* Basic

Syntax

The syntax of a FIDO U2F attestation statement is defined as follows:

```
$$attStmtType = {
  fmt: "fido-u2f",
  attStmt: u2fStmtFormat
}
```

```
u2fStmtFormat = {
  x5c: [ attestnCert: bytes, * (caCert: bytes) ],
  sig: bytes
}
```

The semantics of the above fields are as follows:

```
x5c
```

The elements of this array contain the attestation certificate and its certificate chain, each encoded in X.509 format. The attestation certificate must be the first element in the array.

```
sig
```

The attestation signature.

Signing procedure

If the credential public key of the given credential is not of algorithm “ES256”, stop and return an error.

If the given clientDataHash is 256 bits long, set tbsHash to this value. Otherwise set tbsHash to the SHA-256 hash of the given clientDataHash.

Generate a signature as specified in [FIDO-U2F-Message-Formats] section 4.3, with the application parameter set to the SHA-256 algorithm “ES256”, stop and return an error.

+ Verify that the nonce in the response is identical to the concatenation of the clientDataHash.
+ Verify that the attestation certificate is issued to the hostname "attest.android.com" (see SafetyNet online documentation).
+ Verify that the ctsProfileMatch attribute in the payload of response is true.
+ If successful, return attestation type Basic with the trust path set to the above attestation certificate.

7.6. FIDO U2F Attestation Statement Format

This attestation statement format is used with FIDO U2F authenticators using the formats defined in [FIDO-U2F-Message-Formats].

Attestation statement format identifier

* fido-u2f

Attestation types supported

* Basic

Syntax

The syntax of a FIDO U2F attestation statement is defined as follows:

```
$$attStmtType = {
  fmt: "fido-u2f",
  attStmt: u2fStmtFormat
}
```

```
u2fStmtFormat = {
  x5c: [ attestnCert: bytes, * (caCert: bytes) ],
  sig: bytes
}
```

The semantics of the above fields are as follows:

```
x5c
```

The elements of this array contain the attestation certificate and its certificate chain, each encoded in X.509 format. The attestation certificate must be the first element in the array.

```
sig
```

The attestation signature.

Signing procedure

If the credential public key of the given credential is not of algorithm “ES256”, stop and return an error.

Let authenticatorData denote the authenticator data for the attestation, and let clientDataHash denote the hash of the serialized client data.

If clientDataHash is 256 bits long, set tbsHash to this value. Otherwise set tbsHash to the SHA-256 hash of clientDataHash.

Generate a signature as specified in [FIDO-U2F-Message-Formats] section 4.3, with the application parameter set to the SHA-256 algorithm “ES256”, stop and return an error.

+ Verify that the nonce in the response is identical to the concatenation of the authenticatorData and clientDataHash.
+ Verify that the attestation certificate is issued to the hostname "attest.android.com" (see SafetyNet online documentation).
+ Verify that the ctsProfileMatch attribute in the payload of response is true.
+ If successful, return attestation type Basic with the trust path set to the above attestation certificate.
authenticator arguments are passed as name-value pairs, with the `authenticatorGetAssertion` call (for authentication extensions). These extensions define the following steps and data:

- Generate the claimed to-be-signed data as specified in [FIDO-U2F-Message-Formats] section 4.3, with the `clientDataHash` parameter set to the claimed credential ID of the given credential, and the user public key parameter set to the claimed credential public key.
- Verify that the sig is a valid ECDSA P-256 signature over the to-be-signed data constructed above.
- If successful, return attestation type Basic with the trust path set to x5c.

8. WebAuthn Extensions

The mechanism for generating scoped credentials, as well as requesting and generating Authentication assertions, as defined in 4 Web Authentication API, can be extended to suit particular use cases. Each case is addressed by defining a registration extension and/or an authentication extension. Extensions can define additions to the following steps and data:

- `makeCredential()` request parameters for registration extension.
- `getAssertion()` request parameters for authentication extensions.
- Client processing, and the `ClientData` structure, for registration extensions and authentication extensions.
- Authenticator processing, and the `authenticatorData` structure, for registration extensions and authentication extensions.

When requesting an assertion for a scoped credential, a 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 client arguments for each extension in the `getAssertion()` call (for authentication 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. In addition, the client collects the authenticator arguments for the above extensions, and passes them to the authenticator in the `authenticatorMakeCredential` call (for registration extensions) or `authenticatorGetAssertion` call (for authentication extensions). These authenticator arguments are passed as name-value pairs, with the

8. WebAuthn Extensions

The mechanism for generating public key credentials, as well as requesting and generating Authentication assertions, as defined in 4 Web Authentication API, can be extended to suit particular use cases. Each case is addressed by defining a registration extension and/or an authentication extension.

Every extension is a client extension, meaning that the extension involves communication with and processing by the client. Client extensions define the following steps and data:

- `navigator.credential.create()` extension request parameters and response values for registration extensions.
- `navigator.credential.get()` call (for authentication extensions) or `create()` call (for registration extensions) to the client platform. The client platform processes client extension processing for each extension that it supports, and augments the client data as specified by each extension, by including the extension identifier and client extension output values.

An extension can also be an authenticator extension, meaning that the extension involves communication with and processing by the
extension identifier as the name, and the corresponding authenticator argument as the value. The authenticator, in turn, performs additional processing for the extensions that it supports, and augments authenticatorData as specified by the extension.

All WebAuthn extensions are optional for both clients and authenticators. Thus, any extensions requested by a 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 WebAuthn API processing, so when Relying Parties include extensions with any API calls, they must be prepared to handle cases where some or all of those extensions are ignored.

Clients wishing to support the widest possible range of extensions may choose to pass through any extensions that they do not recognize to authenticators, generating the authenticator argument by simply encoding the client argument in CBOR. All WebAuthn extensions MUST be defined in such a way that this implementation choice does not endanger the user's security or privacy. For instance, if an extension requires client processing, it could be defined in a manner that ensures such a pass-through will produce a semantically invalid authenticator argument, resulting in the extension being ignored by the authenticator. Since all extensions are optional, this will not cause a functional failure in the API operation.

8.1. Extension Identifiers

Extensions are identified by a string, called an extension identifier, chosen by the extension author.

Extension identifiers SHOULD be registered per [WebAuthn-Registries] "Registries for Web Authentication (WebAuthn)": All registered extension identifiers are unique amongst themselves as a matter of course.

Unregistered extension identifiers should aim to be globally unique, e.g., by including the defining entity such as myCompany_extension.

All extension identifiers MUST be a maximum of 32 octets in length and MUST consist only of printable USASCII characters, i.e., VCHAR as defined in [RFC5234]. Implementations MUST match WebAuthn extension identifiers in a case-insensitive fashion.

Authenticator. Authenticator extensions define the following steps and data:
- authenticatorMakeCredential extension request parameters and response values for registration extensions.
- authenticatorGetAssertion extension request parameters and response values for authentication extensions.
- Authenticator extension processing for registration extensions and authentication extensions.

For authenticator extensions, as part of the client extension processing, the client also creates the CBOR authenticator extension input value for each extension (often based on the corresponding client extension input value), and passes them to the authenticator in the create() call (for registration extensions) or the get() call (for authentication extensions). These authenticator extension input values are represented in CBOR and passed as name-value pairs, with the extension identifier as the name, and the corresponding authenticator extension input as the value. The authenticator, in turn, performs additional processing for the extensions that it supports, and returns the CBOR authenticator extension output for each as specified by the extension.

Part of the client extension processing for authenticator extensions is to use the authenticator extension output as an input to creating the client extension output.

All WebAuthn extensions are optional for both clients and authenticators. Thus, any extensions requested by a 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 WebAuthn API processing, so when Relying Parties include extensions with any API calls, they must be prepared to handle cases where some or all of those extensions are ignored.

Clients wishing to support the widest possible range of extensions may choose to pass through any extensions that they do not recognize to authenticators, generating the authenticator extension input by simply encoding the client extension input in CBOR. All WebAuthn extensions MUST be defined in such a way that this implementation choice does not endanger the user's security or privacy. For instance, if an extension requires client processing, it could be defined in a manner that ensures such a pass-through will produce a semantically invalid authenticator extension input value, resulting in the extension being ignored by the authenticator. Since all extensions are optional, this will not cause a functional failure in the API operation. Likewise, clients can choose to produce a client extension output value for an extension that does not understand by encoding the authenticator extension input value into JSON, provided that the CBOR output uses only types present in JSON.

The IANA "WebAuthn Extension Identifier" registry established by [WebAuthn-Registries] should be consulted for an up-to-date list of registered WebAuthn Extensions.

8.1. Extension Identifiers

Extensions are identified by a string, called an extension identifier, chosen by the extension author.

Extension identifiers SHOULD be registered per [WebAuthn-Registries] "Registries for Web Authentication (WebAuthn)": All registered extension identifiers are unique amongst themselves as a matter of course.

Unregistered extension identifiers should aim to be globally unique, e.g., by including the defining entity such as myCompany_extension.

All extension identifiers MUST be a maximum of 32 octets in length and MUST consist only of printable USASCII characters, excluding backslash and doublequote, i.e., VCHAR as defined in [RFC5234] but without %x22 and %x26. Implementations MUST match WebAuthn extension identifiers in
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, e.g., myCompany_extension_01

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.

9 Pre-defined extensions define an initial set of currently-defined and registered extensions. See the WebAuthn Extension Identifiers Registry defined in [WebAuthn-Registries] for an up-to-date list of registered WebAuthn Extension Identifiers.

8.2. 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. Additionally, extensions may specify additional values in ClientData, authenticatorData (in the case of authentication extensions), or both. Finally, if the extension requires any authenticator processing, it must also specify an authenticator argument to be sent via the authenticatorGetAssertion or authenticatorMakeCredential call.

Any extension that requires client processing MUST specify a method of augmenting ClientData that unambiguously lets the Relying Party know that the extension was honored by the client. Similarly, any extension that requires authenticator processing MUST specify a method of augmenting authenticatorData. These methods of augmenting ClientData and authenticatorData are thus treated as different extensions, e.g., myCompany_extension_01

8.3. Extending request parameters

An extension defines up to two request arguments. The client argument is passed from the 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.

A Relying Party simultaneously requests the use of an extension and sets its client argument by including an entry in the extensions option to the create() call. The entry key MUST be the extension identifier, and the value MUST be the client argument. var assertionPromise = credentials.getAssertion(..., { /* extensions */});

"webauthnExample_fooBar": 42

}});

Extension definitions MUST specify the valid values for their client argument. Clients SHOULD ignore extensions with an invalid client argument, set to true to signify that the extension is requested by the Relying Party.
Extensions that only affect client processing need not specify an authenticator argument. Extensions that affect authenticator processing MUST specify a method of computing the authenticator argument from the client argument. For extensions that do not require additional parameters, and are defined as taking a Boolean client argument set to true, this method SHOULD consist of passing an authenticator argument of true (CBOR major type 7, value 21).

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.

8.4. 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 Relying Party to verify the processing took place, or if the processing has a result value that the Relying Party needs to be aware of, the extension should specify a client data value to be included in the ClientData structure.

Each extension, the client SHOULD add an entry to this dictionary with the extension identifier as the key, and the extension's client data value.

Extensions that require authenticator processing MUST define the process by which the client argument can be used to determine the authenticator argument.

8.5. Extending authenticator processing

Extensions that define additional authenticator processing may similarly define an authenticator data value. The value may be any data that can be encoded in CBOR. An authenticator extension that defines such a value must include it in the authenticatorData.

As specified in 5.1 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.

8.6. 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 assertions.

The extension identifier is chosen as webauthnExample_geo. The client argument is the constant value true, since the extension does not require the Relying Party to pass any particular information to the client, other than that it requests the use of the extension. The Relying Party sets this value in its request for an assertion.
The extension also requires the client to set the authenticator parameter to the fixed value true.

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]):

```
FA C1 5F E3 7F                      -- Element 2: Longitude as CBOR encoded float
FA 42 82 1E B3                      -- Element 1: Latitude as CBOR encoded float
FA C1 5F E3 7F                      -- Element 2: Longitude as CBOR encoded float
```

```
81 (hex) -- Flags, ED and TUP both set.
20 05 58 1F -- Signature counter
A1 -- CBOR map of one element
73 -- Key 1: CBOR text string of 19 byt
es
```

```
77 65 62 61 75 74 68 6E 45 78 61 73                                      -- Key 1: CBOR text string of 19 byte
```

```
6D 70 6C 65 5F 67 65 6F             -- "webauthnExample_geo" [=UTF-8 encoded string]
```

```
82 -- Value 1: CBOR array of two elements
```

```
FA 42 82 1E B3                      -- Element 1: Latitude as CBOR encoded float
FA C1 5F E3 7F                      -- Element 2: Longitude as CBOR encoded float
```

```
expressionPromise = navigator.credentials.get({
  publicKey: {
    challenge: "SGFuIFNvbG8gc2hvdCBmaXZjdC4",
    allowList: [], /* Empty filter */
    extensions: { webauthnExample_geo: true }
  }
});
```

This section defines an initial set of extensions. These are recommended for implementation by user agents targeting broad interoperability.

9.1. FIDO App Id

This authentication extension allows Relying Parties that have previously registered a credential using the legacy FIDO JavaScript APIs to request an assertion. Specifically, this extension allows

```
var assertionPromise = credentials.getAssertion("SGFuIFNvbG8gc2hvdCBmaXZjdC4",
  { "webauthnExample_geo": true });
```

This authentication extension allows Relying Parties that have previously registered a credential using the legacy FIDO JavaScript APIs to request an assertion. Specifically, this extension allows

```
var assertionPromise = navigator.credentials.get({
  publicKey: {
    challenge: "SGFuIFNvbG8gc2hvdCBmaXZjdC4",
    allowList: [], /* Empty filter */
    extensions: { webauthnExample_geo: true }
  }
});
```

This section defines an initial set of extensions to be registered in the IANA "WebAuthn Extension Identifier" registry established by [WebAuthn-Registries]. These are recommended for implementation by user agents targeting broad interoperability.

9.1. FIDO AppId Extension (appid)
3. The client argument encoded as a CBOR text string (major type 3).

Extension identifier

fido_appid

Client argument

A single UTF-8 encoded string specifying a FIDO appId.

Client processing

If rpId is present, reject promise with a DOMException whose name is "NotAllowedError", and terminate this algorithm. Replace the calculation of rpId in Step 3 of 4.1.2 Use an existing credential - getAssertion() method with the following procedure: The client uses the value of fido_appid to perform the AppId validation procedure (as defined by [FIDO-APPID]). If valid, the value of rpId for all client processing should be replaced by the value of fido_appid.

Authenticator argument

none

Authenticator processing


9.2. Transaction authorization

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

Extension identifier

webauthn_txAuthSimple

Client argument

A single UTF-8 encoded string prompt.

Client processing

None, except default forwarding of client argument to authenticator argument.

Authenticator argument


5.0/86
9.3. Authenticator Selection Extension

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

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 encoded 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_txAuthGeneric

Client argument

A CBOR map defined as follows:

```
{ contentType: text,   ; MIME-Type of the content, e.g.
  "image/png"    ; content: bytes }
```

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.

This registration extension and authentication extension allows images to be used as transaction authorization prompts as well. This allows authenticators without a font rendering engine to be used and also supports a richer visual appearance.

Extension identifier

webauthn_txAuthGeneric

Client extension input

A CBOR map defined as follows:

```
{ contentType: text,   ; MIME-Type of the content, e.g.
  "image/png"    ; content: bytes }
```

None, except creating the authenticator extension input from the client extension input.

Client extension output

Returns the base64url encoding of the authenticator extension output value as a JSON string

Authenticator extension input

The client extension input encoded as a CBOR text string (major type 3).

Authenticator extension processing

The authenticator MUST display the prompt to the user before performing either user verification or test of user presence. The authenticator may insert line breaks if needed.

Authenticator extension output

A single CBOR string, representing the prompt as displayed (including any eventual line breaks).

9.4. Authenticator Selection Extension (authnSel)

This registration extension allows a Relying Party to guide the selection of the authenticator that will be leveraged when creating the credential. It is intended primarily for Relying Parties that wish to
9.4. Supported Extensions Extension

webauthn_authnSel

Client argument

A sequence of AAGUIDs:

typedef sequence<AAGUID> AuthenticatorSelectionList;

Each AAGUID corresponds to an authenticator model that is acceptable to the 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 model 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.

9.5. Supported Extensions Extension (exts)

webauthn_exts

Client argument

The Boolean value true to indicate that this extension is requested by the Relying Party.

Client processing

None, except default forwarding of client argument to authenticator argument.

Authenticator argument

The Boolean value true, encoded in CBOR (major type 7, value 21).
9.5 User Verification Index (UVI) Extension

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 authenticator data containing one UVI extension:

---
A1
-- TUP and ED set
0 0 0 0 01
-- all public key alg etc.
6C
-- Key 1: CBOR text string of 11 bytes
77 65 62 61 75 74 66 5F 75 76 69 -- "webauthn_uvi" UTF-8 encoded string
58 20
-- Value 1: CBOR byte string with 0x20 bytes
0 0 43 48 45 46 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
B2

9.6. Location Extension

Extension identifier

webauthn_loc

Client argument

The Boolean value true to indicate that this extension is requested by the Relying Party.

Client processing

None, except default forwarding of client argument to authenticator argument.

Authenticator argument

The Boolean value true, encoded in CBOR (major type 7, value 21).

Authenticator processing

If the authenticator does not support the extension, then the authenticator MUST ignore the extension request. If the authenticator accepts the extension, then the authenticator SHOULD only add this extension data to a packed attestation or assertion.

Authenticator data

If the authenticator accepts the extension request, then authenticator data SHOULD provide location data in the form of a CBOR-encoded map, with the first value being the extension identifier and the second being an array of returned values. The array elements SHOULD be derived from (key,value) pairings for each location attribute that the authenticator supports. The following is an example of authenticator data where the returned array is comprised of a (longitude, latitude, altitude) triplet, following the coordinate representation defined in The W3C.

---
A1
-- extension: CBOR map of one element
6C
-- Key 1: CBOR text string of 11 bytes
77 65 62 61 75 74 66 5F 75 76 69 -- "webauthn_uvi" UTF-8 encoded string
58 20
-- Value 1: CBOR byte string with 0x20 bytes
0 0 43 48 45 46 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
B2

9.7. Location Extension (loc)

The location registration extension and authentication extension provides the client device's current location to the WebAuthn relying party.

Extension identifier

loc

Client extension input

The Boolean value true to indicate that this extension is requested by the Relying Party.

Client extension processing

None, except creating the authenticator extension input from the client extension input.

Client extension output

Returns a JSON object that encodes the location information in the authenticator extension output as a Coordinates value, as defined by The W3C Geolocation API Specification.

Authenticator extension input

The Boolean value true, encoded in CBOR (major type 7, value 21).

Authenticator extension processing

If the authenticator does not support the extension, then the authenticator MUST ignore the extension request. If the authenticator accepts the extension, then the authenticator SHOULD only add this extension data to a packed attestation or assertion.

Authenticator extension output

If the authenticator accepts the extension request, then authenticator extension output SHOULD provide location data in the form of a CBOR-encoded map, with the first value being the extension identifier and the second being an array of returned values. The array elements SHOULD be derived from (key,value) pairings for each location attribute that the authenticator supports. The following is an example of authenticator data where the returned array is comprised of a (longitude, latitude, altitude) triplet, following the coordinate representation.
Geolocation API Specification.

The W3C Geolocation API Specification.

The Boolean value true to indicate that this extension is requested by the WebAuthn Relying Party.

Client processing

- The Boolean value true, encoded in CBOR (major type 7, value 21).

Authenticator processing

- The authenticator augments the authenticator data with a user verification index indicating the method used by the user to authorize the operation, as defined below. This extension can be added to attestation objects and assertions.

Authenticator data

- Authenticators can report up to 3 different user verification methods (factors) used in a single authentication instance, using the CBOR syntax defined below:
The semantics of the fields in each uvmEntry are as follows:

userVerificationMethod
The authentication method/factor used by the authenticator to verify the user. Available values are defined in [FIDOReg], "User Verification Methods" section.

keyProtectionType
The method used by the authenticator to protect the FIDO registration private key material. Available values are defined in [FIDOReg], "Key Protection Types" section.

Example for authenticatorData containing one UVM extension for a multi-factor authentication instance where 2 factors were used:

```
# -- RP ID hash (32 bytes)
81 00 00 00 01 -- TUP and ED set
A1 -- (initial) signature counter
6C 77 65 62 61 75 74 68 6E 2E 75 76 6d -- "webauthn_uvm" UTF-8 encoded string
82 83 usage -- Value 1: CBOR array of length 2 indicating two factor
02 -- Subitem 1: CBOR array of length 3
04 -- Subitem 2: CBOR short for User Verification Method
08 -- Subitem 3: CBOR short for Matcher Protection Type TEE
00 00 00 01 Passcode re -- Subitem 2: CBOR short for Key Protection Type Software
01 -- Subitem 3: CBOR short for Matcher Protection Type Software
```

This specification registers the algorithm names "S256", "S384", "SS12", 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-sp-15.

Algorithm Name: "S256"
This section registers the attestation statement formats defined in Section 7 Defined Attestation Statement Formats in the IANA "WebAuthn Attestation Statement Format Identifier" registry established by [WebAuthn-Registries].

- **WebAuthn Attestation Statement Format Identifier**: packed
  - Description: The "packed" attestation statement format is a WebAuthn-optimized format for attestation data. It uses a very compact but still extensible encoding method. This format is implementable by authenticators with limited resources (e.g., secure elements).
  - Specification Document: Section 7.2 Packed Attestation Statement Format of this specification
  - WebAuthn Attestation Statement Format Identifier: `tpm`
  - Description: The TPM attestation statement format returns an attestation statement in the same format as the packed attestation statement format, although the the rawData and signature fields are computed differently.
  - Specification Document: Section 7.3 TPM Attestation Statement Format of this specification

- **WebAuthn Extension Identifier Registrations**

**Section 10.2. WebAuthn Extension Identifier Registrations**

This section registers the extension identifier values defined in Section 8 WebAuthn Extensions in the IANA "WebAuthn Extension Identifier" registry established by [WebAuthn-Registries].

- **WebAuthn Extension Identifier**: appid
  - Description: This authentication extension allows Relying Parties that have previously registered a credential using the legacy FIDO JavaScript APIs to request an assertion.
  - Specification Document: Section 9.1 FIDO AppId Extension (appid) of this specification
  - WebAuthn Extension Identifier: txAuthSimple
  - Description: This registration extension and authentication 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.
  - Specification Document: Section 9.2 Simple Transaction Authorization Extension (txAuthSimple) of this specification
  - WebAuthn Extension Identifier: txAuthGeneric
  - Description: This registration extension and authentication extension allows images to be used as transaction authorization
the authenticator, and the authenticator needs to allow the client
platform needs to display any prompts that would otherwise be shown by
to specific implementation considerations. Specifically, the client
in the client platform. The flow also works for the case of an
without modification for the case of an authenticator that is embedded
authenticator types are also supported by this API, subject to
One example of such an authenticator would be a smart phone. Other
involving an external first-factor authenticator with its own display.
As was the case in earlier sections, this flow focuses on a use case
in this API. Note that this is an example flow, and does not limit the
This section is not normative.
In this section, we walk through some events in the lifecycle of a
one scoped credential, along with the corresponding sample code for using
This section is not normative.
In this section, we walk through some events in the lifecycle of a
One example of such an authenticator would be a smart phone. Other
without modification for the case of an authenticator that is embedded
in the client platform. The flow also works for the case of an
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
prompts as well. This allows authenticators without a font
rendering engine to be used and also supports a richer visual
appearance than accomplished with the webauthn.txauth.simple
extension.
* Specification Document: Section 9.3 Generic Transaction
* Authorization Extension (txAuthGeneric) of this specification
* WebAuthn Extension Identifier: authnSel
* Description: 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.
* Specification Document: Section 9.4 Authenticator Selection
* Extension (authnSel) of this specification
  * WebAuthn Extension Identifier: exts
  * Description: This registration extension enables the Relying Party
to determine which extensions the authenticator supports. The
extension data is a list (CBOR array) of extension identifiers
encoded as UTF-8 Strings. This extension is added automatically by
the authenticator. This extension can be added to attestation
statements.
* Specification Document: Section 9.5 Supported Extensions Extension
  (exts) of this specification
  * WebAuthn Extension Identifier: uvi
  * Description: This registration extension and authentication
  extension enables use of a user verification index. The user
  verification index is a value uniquely identifying a user
  verification data record. 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".
  * Specification Document: Section 9.6 User Verification Index
  Extension (uvi) of this specification
  * WebAuthn Extension Identifier: loc
  * Description: This location registration extension and authentication
  extension provides the client device's current location to the
  WebAuthn relying party, if supported by the client device and
  subject to user consent.
  * Specification Document: Section 9.7 Location Extension (loc) of
  this specification
  * WebAuthn Extension Identifier: uvm
  * Description: This registration extension and authentication
  extension enables use of a user verification method. The user
  verification method extension returns to the Webauthn relying party
  which user verification methods (factors) were used for the
  WebAuthn operation.
  * Specification Document: Section 9.8 User Verification Method
  Extension (uvm) of this specification

11. Sample scenarios

This section is not normative.

In this section, we walk through some events in the lifecycle of a
public key 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
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
11.1. Registration

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

1. The user visits example.com, which serves up a script. At this point, the user must already be logged in using a legacy username and password, or additional authenticator, or other means acceptable to the Relying Party.

2. The Relying Party script runs the code snippet below.

3. The client platform searches for and locates the authenticator.

4. The client platform connects to the authenticator, performing any pairing actions if necessary.

5. The 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 authenticator returns a response to the client platform, which in turn returns a response to the 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 Relying Party script sends the newly generated credential public key to the server, along with additional information such as attestation regarding the provenance and characteristics of the authenticator.
   + The server stores the credential public key in its database and associates it with the user as well as with the characteristics 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:

```javascript
var webauthnAPI = navigator.authentication;

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

var userAccountInformation = {
  rp: {
    // Relying Party:
    rpName: "Acme",
    icon: "https://pics.acme.com/00/p/aBjjjpqPb.png",
    displayNames: ["John P. Smith", "johnp.smith@example.com"],
    name: "johnp.smith@example.com",
    imageURL: "https://pics.acme.com/00/p/aBjjjpqPb.png"
  },

  // User:
  user: {
    id: "1098237235409872",
    name: "John P. Smith",
    displayNames: ["John P. Smith", "johnp.smith@example.com"],
    icon: "https://pics.acme.com/00/p/aBjjjpqPb.png"
  }
};

// This Relying Party will accept either an ES256 or RS256 credential, but prefers an ES256 credential.
var cryptoParams = [
  {
    type: "ScopedCred",
    algorithm: "ES256"
  },
  {
    type: "ScopedCred",
    algorithm: "RS256"
  }
];

var challenge = new TextEncoder().encode("climb a mountain");
```

1. The user visits example.com, which serves up a script at this point, the user must already be logged in using a legacy username and password, or additional authenticator, or other means acceptable to the Relying Party.

2. The Relying Party script runs the code snippet below.

3. The client platform searches for and locates the authenticator.

4. The client platform connects to the authenticator, performing any pairing actions if necessary.

5. The 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 authenticator returns a response to the client platform, which in turn returns a response to the 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 Relying Party script sends the newly generated credential public key to the server, along with additional information such as attestation regarding the provenance and characteristics of the authenticator.
   + The server stores the credential public key in its database and associates it with the user as well as with the characteristics 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:

```javascript
if (!PublicKeyCredential) { /* Platform not capable. Handle error. */ }

var publicKey = {
  challenge: Uint8Array.from(window.atob("PGifxAoBwCkWkm4b1CiIl5otCphiIh6MijdjbWFjomA=")),
  c: window.crypto hashlibwards("FjomA="),
  c: window.crypto charCodeAt(0)
};

// Relying Party:
var publicKey = {
  // Relying Party:
  parameters: [
    {
      type: "public-key",
      algorithm: "ES256"
    },
    {
      type: "public-key",
      algorithm: "RS256"
    }
  ],

  // This Relying Party will accept either an ES256 or RS256 credential, but prefers an ES256 credential.
  parameters: [
    {
      type: "public-key",
      algorithm: "ES256"
    },
    {
      type: "public-key",
      algorithm: "RS256"
    }
  ],

  timeout: 60000, // 1 minute

  // This Relying Party will accept either an ES256 or RS256 credential, but prefers an ES256 credential.
  parameters: [
    {
      type: "public-key",
      algorithm: "ES256"
    },
    {
      type: "public-key",
      algorithm: "RS256"
    }
  ],

  // This Relying Party will accept either an ES256 or RS256 credential, but prefers an ES256 credential.
  parameters: [
    {
      type: "public-key",
      algorithm: "ES256"
    },
    {
      type: "public-key",
      algorithm: "RS256"
    }
  ],

  timeout: 60000, // 1 minute
```
allowList: [{ type: "ScopedCred" }]

var options = {
  timeout: 60000, // 1 minute
  excludeList: []
}; // No excludeList

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

var challenge = new TextEncoder().encode("climb a mountain");

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

webauthnAPI.makeCredential(userAccountInformation, cryptoFarms, challenge, options);
.then(function(newCredentialInfo) {
  // Send new credential info to server for verification and registration.
});.catch(function(err) {
  // No acceptable authenticator or user refused consent. Handle appropriately
});;

11.2. 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 an Authentication
   Assertion, providing as much information as possible to narrow the
   choice of acceptable credentials for the user. This may be obtained
   from the data that was stored locally after registration, or by
   other means such as prompting the user for a username.
3. The Relying Party script runs one of the code snippets below.
4. The client platform searches for and locates the authenticator.
5. The client platform connects to the authenticator, performing any
   pairing actions if necessary.
6. The 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 authenticator returns a response to the client platform, which
   in turn returns a response to the 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, extracts the credential ID,
     looks up the registered credential public key it is database,
     and verifies the assertion's authentication signature. If valid,
     it looks up the identity associated with the assertion's credential ID.
     If the credential ID is not recognized by the server (e.g., it
     has been deregistered due to inactivity) then the
     authentication has failed; each 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 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:

var webauthnAPI = navigator.authentication;

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

var challenge = new TextEncoder().encode("climb a mountain");
var options = {
  timeout: 60000, // 1 minute
  allowList: [{ type: "ScopedCred" }]
}; // No allowList

webauthnAPI.makeCredential(userAccountInformation, cryptoFarms, challenge, options);
.then(function(newCredentialInfo) {
  // Send new credential info to server for verification and registration.
});.catch(function(err) {
  // No acceptable authenticator or user refused consent. Handle appropriately
});;

11.2. 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 an Authentication
   Assertion, providing as much information as possible to narrow the
   choice of acceptable credentials for the user. This may be obtained
   from the data that was stored locally after registration, or by
   other means such as prompting the user for a username.
3. The Relying Party script runs one of the code snippets below.
4. The client platform searches for and locates the authenticator.
5. The client platform connects to the authenticator, performing any
   pairing actions if necessary.
6. The 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 authenticator returns a response to the client platform, which
   in turn returns a response to the 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, extracts the credential ID,
     looks up the registered credential public key it is database,
     and verifies the assertion's authentication signature. If valid,
     it looks up the identity associated with the assertion's credential ID.
     If the credential ID is not recognized by the server (e.g., it
     has been deregistered due to inactivity) then the
     authentication has failed; each 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 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:

var options = {
  challenge: new TextEncoder().encode("climb a mountain"),
  timeout: 60000, // 1 minute
  allowList: [{ type: "public-key" }]
}; // No allowList

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

var options = {
  excludeList: []
}; // No excludeList

webauthnAPI.makeCredential(userAccountInformation, cryptoFarms, challenge, options);
.then(function(newCredentialInfo) {
  // Send new credential info to server for verification and registration.
});.catch(function(err) {
  // No acceptable authenticator or user refused consent. Handle appropriately
});;
On the other hand, if the Relying Party script has some hints to help it narrow the list of credentials, then the sample code for performing such an authentication might look like the following. Note that this sample also demonstrates how to use the extension for transaction authorization.

```javascript
if (webauthnAPI) { /* Platform not capable. Handle error. */
  var encoder = new TextEncoder();
  var challenge = encoder.encode("climb a mountain");

  var acceptableCredential1 = {
    type: "ScopedCred",
    id: encoder.encode("!!!!!!!Hi there!!!\n")
  }

  var acceptableCredential2 = {
    type: "ScopedCred",
    id: encoder.encode("roses are red, violets are blue\n")
  }

  var options = {
    timeout: 60000, // 1 minute
    allowList: [acceptableCredential1, acceptableCredential2],
    extensions: ['webauthn.txauth.simple']
  };

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

      // On the other hand, if the Relying Party script has some hints to help it narrow the list of credentials, then the sample code for performing such an authentication might look like the following. Note that this sample also demonstrates how to use the extension for transaction authorization.
      
      if (!webauthnAPI) { /* Platform not capable. Handle error. */
        var encoder = new TextEncoder();
        var acceptableCredential1 = {
          type: "public-key",
          id: encoder.encode("!!!!!Hi there!!\n")
        }
        var acceptableCredential2 = {
          type: "public-key",
          id: encoder.encode("roses are red, violets are blue\n")
        }

        var options = {
          challenge: encoder.encode("climb a mountain"),
          timeout: 60000, // 1 minute
          allowList: [acceptableCredential1, acceptableCredential2],
          extensions: ["webauthn.txauth.simple"]
        };

        navigator.credentials.get({"publicKey":options})
          .then(function (assertion) {
            // Send assertion to server for verification
          })
          .catch(function(err) {
            // No acceptable credential or user refused consent. Handle appropriately.
          });
```

```
11.3. 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 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 future, the Relying Party script does not specify this credential in any list of acceptable credentials, and assertions signed by this credential are rejected.
- Possibility #3 -- user deletes the credential from the device.
  + User employs a device-specific method (e.g., device settings UI) to delete a credential from their device.
```
12. Acknowledgements

We thank the following for their contributions to, and thorough review of, this specification: Domenic Denicola, Rahul Ghosh, Brad Hill, Jing Jin, Anne van Kesteren, Giridhar Mandyam, Axel Nennker, Yaron Sheffer, Mike West, Boris Zbarsky.

Index

Terms defined by this specification

- AAGUID, in 9.3
- Account, in 4.3
- accountInformation, in 4.1.1
- algorithm
- dict-member for ScopedCredentialParameters, in 4.4
- dfn for ScopedCredentialParameters, in 4.4
- AlgorithmIdentifier, in 2.1
- allowList
- dict-member for AssertionOptions, in 4.7
- dfn for AssertionOptions, in 4.7
- ASCII case-sensitive match, in 3
- Assert, in 3
- assertionChallenge, in 4.1.2
- AssertionOptions, in 4.7
- assertion signature, in 5
- Attachment, in 4.5.1
- attachment
- dict-member for ScopedCredentialOptions, in 4.5
- dfn for ScopedCredentialOptions, in 4.5
- Attestation, in 3
- Attestation Certificate, in 3
- AttestationChallenge, in 4.1.1
- Attestation information, in 3
- attestation key pair, in 3
- attestationObject
- attribute for ScopedCredentialInfo, in 4.2
- dfn for ScopedCredentialInfo, in 4.2
- attestation objects, in 3
- attestation private key, in 3
- attestation public key, in 3
- attestation signature, in 3
- attestation statement format, in 5.3
- attestation statement format identifier, in 7.1
- attestation type, in 5.3
- Authentication, in 3
- authentication, in 4
- Authentication Assertion, in 3
- AuthenticationAssertion, in 4.6
- AuthenticationExtensions, in 4.8
- Authenticator, in 3
- authenticator argument, in 8.3
- authenticatorCancel, in 5.2.3
- authenticatorData
- attribute for AuthenticationAssertion, in 4.6
- definition of, in 5.1

+ 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.

12. Acknowledgements

We thank the following for their contributions to, and thorough review of, this specification: Richard Barnes, Dominic Battr, Domenic Denicola, Rahul Ghosh, Brad Hill, Jing Jin, Angela Liao, Anne van Kesteren, Ian Kilpatrick, Giridhar Mandyam, Axel Nennker, Kimberly Paulhamus, Adam Powers, Yaron Sheffer, Mike West, Jeffrey Yasskin, Boris Zbarsky.

Index

Terms defined by this specification

- AAGUID, in 9.4
- algorithm, in 4.3
- allowList, in 4.8
- assertion, in 3
- assertion signature, in 5
- Attachment, in 4.5.3
- attachment, in 4.5.2
- attachment modality, in 4.5.3
- Attestation, in 3
- Attestation Certificate, in 3
- Attestation data, in 5.3.1
- Attestation information, in 3
- attribute for attestationObject, in 4.2.1
- attribute for attestationSignature, in 4.2.1
- attestation objects, in 3
- attestation private key, in 3
- attestation public key, in 3
- attestation signature, in 3
- attestation statement format, in 5.3
- attestation statement format identifier, in 7.1
- attestation type, in 5.3
- Authentication, in 3
- Authentication Assertion, in 3
- authentication extension, in 8
- AuthenticationExtensions
- definition of, in 4.7
- Authenticator, in 3
- AuthenticatorAssertionResponse, in 4.2.2
- AuthenticatorAttestationResponse, in 4.2.1
- authenticatorCancel, in 5.2.3
- authenticator data, in 5.1
- authenticatorData, in 4.2.2
- authenticator data claimed to have been used for the attestation, in 5.3.2
- authenticator extension, in 8
- authenticator extension input, in 8.3
+ dfn for Account, in 4.3
+ attribute for ScopedCredential, in 4.9.3
+ dfn for ScopedCredential, in 4.9.3
+ dict-member for ScopedCredentialDescriptor, in 4.9.4
+ dfn for ScopedCredentialDescriptor, in 4.9.4
  + imageURL
  + dict-member for Account, in 4.3
+ dfn for Account, in 4.3
* makeCredential(accountInformation, cryptoParameters, attestationChallenge), in 4.1
* makeCredential(accountInformation, cryptoParameters, attestationChallenge, options), in 4.1
* name
+ dict-member for Account, in 4.3
+ dfn for Account, in 4.3
  + nfc
  + enum-value for Transport, in 4.9.5
  + dfn for Transport, in 4.9.5
  + "nfc", in 4.9.5
  + options
    + dfn for makeCredential(), in 4.1.1
    + dfn for getAssertion(), in 4.1.2
  + origin
    + dict-member for ClientData, in 4.9.1
    + dfn for ClientData, in 4.9.1
    + platform, in 4.5.1
      + "platform", in 4.5.1
      + platform attachment, in 4.5.1
    + platform authenticators, in 4.5.1

* Privacy CA, in 5.3.3
* Promises, in 2.1

* Registration, in 3
* Relying Party, in 3
* Relying Party Identifier, in 3
* roaming authenticators, in 4.5.1
* rpDisplayName
  + dict-member for Account, in 4.3
  + dfn for Account, in 4.3
+ nfc for Account, in 4.3
+ dfn for Account, in 4.3
+ dfn for makeCredential(), in 4.1.1
+ dfn for getAssertion(), in 4.1.2
+ origin
  + dict-member for ClientData, in 4.9.1
  + dfn for ClientData, in 4.9.1
  + platform, in 4.5.1
    + "platform", in 4.5.1
    + platform attachment, in 4.5.1
  + platform authenticators, in 4.5.1

* RP ID, in 3
* ScopedCred
  + enum-value for ScopedCredentialType, in 4.9.2
  + dfn for ScopedCredentialType, in 4.9.2
  + "ScopedCred", in 4.9.2
  + Scoped Credential, in 3
  + ScopedCredential, in 4.9.3
  +ScopedCredentialDescriptor, in 4.9.4
  +ScopedCredentialInfo, in 4.2
  +ScopedCredentialOptions, in 4.5
  +ScopedCredentialParameters, in 4.4
  +ScopedCredentialType, in 4.9.2
  +secure contexts, in 3
* Self Attestation, in 5.3.3

+ dict-member for PublicKeyCredentialDescriptor, in 4.8.3
  + [Identifier], in 4.1
  + Identifier of the ECDAA-Issuer public key, in 7.2
  + JSON-serialized client data, in 4.8.1
  + MakeCredentialOptions, in 4.5
  + name, in 4.5.1
  + nfc, in 4.8.4
  + origin, in 4.8.1
  + parameters, in 4.5
  + platform, in 4.5.3
  + "platform", in 4.5.3
  + platform attachment, in 4.5.3
  + platform authenticators, in 4.5.3

* Privacy CA, in 5.3.3
* publicKey
  + dict-member for CredentialRequestOptions, in 4.1.1
  + dict-member for CredentialCreationOptions, in 4.1.2
  + publicKey, in 4.8.2
  + PublicKey Credential, in 3
  + PublicKeyCredential, in 4.1
  + PublicKeyCredentialDescriptor, in 4.8.3
  + PublicKeyCredentialEntity, in 4.5.1
  + PublicKeyCredentialParameters, in 4.3
  + PublicKeyCredentialRequestOptions, in 4.6
  + PublicKeyCredentialType, in 4.8.2
  + PublicKeyCredentialUserEntity, in 4.4
  + rawId, in 4.1
  + Registration, in 3
  + registration extension, in 8
  + Relying Party, in 3
  + Relying Party Identifier, in 3
  + requireResidentKey, in 4.5.2
  + response, in 4.1
  + roaming authenticators, in 4.5.3
  + rp, in 4.3
+ rpl, in 4.6

* RP ID, in 3
* Self Attestation, in 5.3.3
**signature**, attribute for AuthenticationAssertion, in 4.6
+ dfn for AuthenticationAssertion, in 4.6
+ timeout
+ dict-member for ScopedCredentialOptions, in 4.5
+ dfn for ScopedCredentialOptions, in 4.5
+ dfn for AssertionOptions, in 4.7
+ dfn for AssertionOptions, in 4.7
+ tokenBinding
+ dict-member for ClientData, in 4.9.1
+ dfn for ClientData, in 4.9.1
+ Transport, in 4.9.5
+ transports, in 4.9.4

* type
+ dict-member for ScopedCredentialParameters, in 4.4
+ dfn for ScopedCredentialParameters, in 4.4
+ attribute for ScopedCredential, in 4.9.3
+ dict-member for ScopedCredentialDescriptor, in 4.9.4
+ dfn for ScopedCredentialDescriptor, in 4.9.4
* "usb", in 4.9.5
* usb
+ enum-value for Transport, in 4.9.5
+ dfn for Transport, in 4.9.5
+ User Consent, in 3
+ User Verification, in 3
+ WebAuthentication, in 4.1
+ Web Authentication API, in 4
+ WebAuthn Client, in 3

Terms defined by reference
* [WHATWG-ENCODING] defines the following terms:
  + utf-8 encoded

* [HTML] defines the following terms:
  + Navigator
* [HTML5] defines the following terms:
  + current settings object
  + navigator
  + opaque origin
  + origin
  + relaxing the same-origin restriction

* signature, in 4.2.2
  + Test of User Presence, in 3
+ timeout
+ dict-member for MakeCredentialOptions, in 4.5
+ dfn for MakeCredentialOptions, in 4.5
+ tokenBinding, in 4.8.1
+ Transport, in 4.8.4
+ transports, in 4.8.3
+ TUP, in 3
+ [[type]], in 4.1

* type
+ dict-member for PublicKeyCredentialParameters, in 4.3
+ dfn for PublicKeyCredentialParameters, in 4.3
+ dfn for PublicKeyCredentialDescriptor, in 4.8.3
+ user, in 4.8.4

* User Consent, in 3
* User Verification, in 3
* User Verified, in 3
* Web Authentication API, in 4
* WebAuthn Client, in 3

Terms defined by reference
* [CREDENTIAL-MANAGEMENT-1] defines the following terms:
  + CredentialCreationOptions
    + create()
  + [ECMAScript] defines the following terms:
    + "arraybuffer"
    + internal slot
    + toString
  + [ENCODING] defines the following terms:
    + utf-8 encode
* [HTML] defines the following terms:
  + ascii serialization of an origin
  + dom manipulation task source
  + effective domain
  + global object
  + in parallel
  + is a registrable domain suffix of or is equal to
  + is not a registrable domain suffix of and is not equal to
  + origin
  + relevant settings object
  + task
  + task source
  + unicode serialization of an origin
* [HTML5] defines the following terms:
  + opaque origin
* [INFRA] defines the following terms:
  + append (for list)
  + append (for set)
  + continue
  + for each (for list)
  + for each (for map)
  + is empty
  + is not empty
  + item
  + list
  + map
  + ordered set
  + remove
* [WebCryptoAPI] defines the following terms:
  + normalizing an algorithm

* [WebIDL] defines the following terms:
  + BufferSource
  + SecureContext

set
  + present

References
Normative References

[CDDL]

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

References
Normative References

[CDDL]

[CREDENTIAL-MANAGEMENT-1]
Mike West. Credential Management Level 1. URL: https://www.w3.org/TR/credential-management-1/

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

[ECMAScript]

[ENCODING]
Anne van Kesteren. Encoding Standard. Living Standard. URL: https://encoding.spec.whatwg.org/
interface PublicKeyCredential : Credential {
   readonly attribute WebAuthentication authentication;
}

[RFC280]

[RFC5454]

[RFC7159]

[SP800-107r1]

[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-Part1]

[TPMv2-Part2]

[TPMv2-Part3]

[UAFProtocol]

[WebAuthn-Registries]

IDL Index

partial interface Navigator {
   readonly attribute WebAuthentication authentication;
}
interface WebAuthentication {
  Promise<ScopedCredentialInfo> makeCredential(Account accountInformation, sequence<ScopedCredentialParameters> cryptoParameters, optional ScopedCredentialOptions options);
}

Promise<AuthenticationAssertion> getAssertion(BufferSource assertionChallenge, optional AssertionOptions options);

interface ScopedCredentialInfo {
  readonly attribute ArrayBuffer attestationObject;
}

dictionary Account {
  required DOMString rpDisplayName;
  required DOMString displayId;
  required DOMString name;
  DOMString imageURL;
}

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

dictionary ScopedCredentialOptions {
  required USVString rpId;
  unsigned long timeout;
  sequence<ScopedCredentialDescriptor> excludeList = [];
  Attachment attachment;
  AuthenticationExtensions extensions;
}

interface AuthenticatorResponse {
  readonly attribute ArrayBuffer authenticatorData;
  readonly attribute ArrayBuffer signature;
  readonly attribute ArrayBuffer clientDataJSON;
}

dictionary AuthenticatorSelectionCriteria {
  required sequence<PublicKeyCredentialDescriptor> excludeList;
  string icon;
  DOMString id;
}

dictionary PublicKeyCredentialEntity {
  required AlgorithmIdentifier algorithm;
}

dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
  readonly attribute ArrayBuffer authenticatorData;
  required PublicKeyCredentialType type;
  readonly attribute ArrayBuffer attestationObject;
}

dictionary AuthenticatorResponse {
  readonly attribute ArrayBuffer clientDataJSON;
}

dictionary AuthenticatorAttestationResponse : AuthenticatorResponse {
  readonly attribute ArrayBuffer attestationObject;
}

dictionary AuthenticatorAssertionResponse : AuthenticatorResponse {
  readonly attribute ArrayBuffer signature;
}

dictionary PublicKeyCredentialRequestOptions {
  MakeCredentialOptions? publicKey;
}

dictionary CredentialCreationOptions {
  PublicKeyCredentialRequestOptions? publicKey;
  optional MakeCredentialOptions publicKey;
  optional sequence<PublicKeyCredentialParameters> parameters;
}

dictionary PublicKeyCredentialOptions {
  required PublicKeyCredentialParameters parameters;
  required USVString icon;
  DOMString id;
  dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
    DOMString displayName;
  }
}

dictionary PublicKeyCredentialParameters {
  required USVString type;
}

dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
  DOMString displayName;
  DOMString id;
  PublicKeyCredentialType type;
  optional sequence<PublicKeyCredentialParameters> parameters;
}

dictionary MakeCredentialOptions {
  required PublicKeyCredentialUserEntity user;
  optional PublicKeyCredentialCredentialedEntity rp;
  required USVString icon;
}

dictionary AuthenticatorSelectionCriteria {
  attachment attachment;
  boolean requireResidentKey = false;
}
4.9.6. Cryptographic Algorithm Identifier (type)

4.9.1. Client data used in WebAuthn signatures (dictionary)

ScopedCredentialParameters

#dictdef-algorithmidentifier

Referenced in:

4.1.2. Use an existing credential - getAssertion() method (2) (3)

4.1.2. Use an existing credential - getAssertion() method (2) (3)

4.1.2. Use an existing credential - getAssertion() method (2) (3)

4.1.2. Use an existing credential - getAssertion() method (2) (3)

4.1.1. Create a new credential - makeCredential() method (2) (3)

4.1.1. Create a new credential - makeCredential() method (2) (3)

#domexception

Referenced in:

4.1.1. Create a new credential - makeCredential() method (2) (3)

4.1.2. Use an existing credential - getAssertion() method (2) (3)

4.1.2. Use an existing credential - getAssertion() method (2) (3)

4.1.2. Use an existing credential - getAssertion() method (2) (3)

4.1.2. Use an existing credential - getAssertion() method (2) (3)

4.1.3. Create a new credential - PublicKeyCredential's

\[
[[Create]](\text{options}) \text{ method (2)} \]

4.1.3. Create a new credential - PublicKeyCredential's

\[
[[Create]](\text{options}) \text{ method (2)} \]

4.1.3. Create a new credential - PublicKeyCredential's

\[
[[Create]](\text{options}) \text{ method (2)} \]

4.1.3. Create a new credential - PublicKeyCredential's

\[
[[Create]](\text{options}) \text{ method (2)} \]

4.1.4. Use an existing credential - PublicKeyCredential's

\[
[[DiscoverFromExternalSource]](\text{options}) \text{ method (2)} \]

4.1.4. Use an existing credential - PublicKeyCredential's

\[
[[DiscoverFromExternalSource]](\text{options}) \text{ method (2)} \]

4.1.4. Use an existing credential - PublicKeyCredential's

\[
[[DiscoverFromExternalSource]](\text{options}) \text{ method (2)} \]

4.1.4. Use an existing credential - PublicKeyCredential's

\[
[[DiscoverFromExternalSource]](\text{options}) \text{ method (2)} \]

typedef BufferSource AAGUID;

typedef sequence<AAGUID> AuthenticatorSelectionList;

typedef sequence<Transport>   transports;

required BufferSource id;

required PublicKeyCredentialType type;

dictionary PublicKeyCredentialDescriptor {
  required BufferSource id;
  sequence<Transport>   transports;
}

enum Transport {
  "usb",
  "nfc",
  "ble"
};

typedef sequence<AAGUID> AuthenticatorSelectionList;

typedef BufferSource AAGUID;

dictdef-algorithmidentifier

Referenced in:

4.4. Parameters for Credential Generation (dictionary)

ScopedCredentialParameters

#dictdef-clientdata

Referenced in:

4.9.1. Client data used in WebAuthn signatures (dictionary)

ClientData

#dictdef-cryptographicalgorithmidentifier

Referenced in:

4.9.0. Cryptographic Algorithm Identifier (type

#base64url-encoding

Referenced in:

4.1. PublicKeyCredential Interface

4.1.3. Create a new credential - PublicKeyCredential's

\[
[[Create]](\text{options}) \text{ method (2)} \]

4.1.4. Use an existing credential - PublicKeyCredential's

\[
[[DiscoverFromExternalSource]](\text{options}) \text{ method (2)} \]
6.2. Verifying an authentication assertion

#dom-publickeycredential-clientextensionresultsReferenced in:
* 4.1. PublicKeyCredential Interface
* 4.1.3. Create a new credential - PublicKeyCredential's
  \[[Create\](options) method
* 4.1.4. Use an existing credential - PublicKeyCredential::\[[DiscoverFromExternalSource\](options) method
* 6.4. Client extension processing

#dom-publickeycredential-identifiersonslotReferenced in:
* 4.1. PublicKeyCredential Interface (2)
* 4.1.3. Create a new credential - PublicKeyCredential's
  \[[Create\](options) method
* 4.1.4. Use an existing credential - PublicKeyCredential::\[[DiscoverFromExternalSource\](options) method

#dictdef-credentialrequestoptionsReferenced in:
* 4.1.1. CredentialRequestOptions Extension
* 4.1.4. Use an existing credential - PublicKeyCredential::\[[DiscoverFromExternalSource\](options) method

#dom-credentialcreationoptions-publickeyReferenced in:
* 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
  PublicKeyCredential::\[[DiscoverFromExternalSource\](options) method

#authenticatoreventReferenced in:
* 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
  PublicKeyCredential::\[[DiscoverFromExternalSource\](options) method

8.4. Client extension processing

#authenticatoreventresultsReferenced in:
* 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
  PublicKeyCredential::\[[DiscoverFromExternalSource\](options) method

#dom-authenticatoreventresultsReferenced in:
* 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
  PublicKeyCredential::\[[DiscoverFromExternalSource\](options) method

6.1. Registering a new credential

#dom-webauthn-getassertionReferenced in:
* 1. Introduction

* 6.1. Registering a new credential

#dom-webauthn-makerequestReferenced in:
rs-attestationchallenge-options-cryptoparametersReferenced in:
* 4.1.1. Create a new credential - makeCredential() method (2) (3)
(4) (5)

* 4.1.1. Create a new credential - makeCredential() method

#dom-webauthn-makerequestReferenced in:
rs-attestationchallenge-options-cryptoparametersReferenced in:
* 4.1.1. Create a new credential - makeCredential() method (2) (3)
(4) (5) (6) (7)

* 4.1.1. Create a new credential - makeCredential() method

#dom-webauthn-makerequestReferenced in:
rs-attestationchallenge-options-cryptoparametersReferenced in:
* 4.1.1. Create a new credential - makeCredential() method (2) (3)
(4) (5) (6) (7)

* 4.1.1. Create a new credential - makeCredential() method

#authenticatoreventReferenced in:
* 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
  PublicKeyCredential::\[[DiscoverFromExternalSource\](options) method

#authenticatoreventresultsReferenced in:
* 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
  PublicKeyCredential::\[[DiscoverFromExternalSource\](options) method

#dom-authenticatoreventresultsReferenced in:
* 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
  PublicKeyCredential::\[[DiscoverFromExternalSource\](options) method

#authenticatoreventresponseReferenced in:
* 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
  PublicKeyCredential::\[[DiscoverFromExternalSource\](options) method

6.2. Verifying an authentication assertion

* 6.2. Verifying an authentication assertion
4.1. WebAuthentication Interface

#dictdef-accountReferenced in:
* 4.1. WebAuthentication Interface

6.1. Registering a new credential (2)

6. Registering a new credential

#authenticatorassertionresponseReferenced in:
* 3. Terminology
* 4.1. Use an existing credential - getAssertion() method
* 6.2. Verifying an authentication assertion
* 7. WebAuthn Extensions (2)
* 8. WebAuthn Extensions (2)
* 8.2. Defining extensions
* 8.3. Extending request parameters (2)
* 9.1. FIDO AppId

#dom-authenticatorattestationresponse-attestationobjectReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's

\[
\text{PublicKeyCredential::}\text{[[Create]](options) method (2)
\]

#dom-authenticatorattestationresponse-authenticatordataReferenced in:
* 4.1.4. Use an existing credential - PublicKeyCredential::\text{[[DiscoverFromExternalSource]](options) method (2)

#dom-authenticatorattestationresponse-signatureReferenced in:
* 4.1.4. Use an existing credential - PublicKeyCredential::\text{[[DiscoverFromExternalSource]](options) method (2)

#dictdef-authenticatorinformationReferenced in:
* 4.1. WebAuthentication Interface

#dictdef-publickeycredentialparametersReferenced in:
* 4.3. Parameters for Credential Generation (dictionary PublicKeyCredentialParameters)

#dictdef-publickeycredentialparameters-typeReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's

\[
\text{PublicKeyCredential::}\text{[[Create]](options) method (2)
\]

\[
\text{PublicKeyCredential::}\text{[[DiscoverFromExternalSource]](options) method (2)
\]

#dictdef-publickeycredentialuserentity-displaynameReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's

\[
\text{PublicKeyCredential::}\text{[[Create]](options) method (2)
\]

\[
\text{PublicKeyCredential::}\text{[[DiscoverFromExternalSource]](options) method (2)
\]
4.5.1. Credential Attachment enumeration (enum Attachment)

ScopedCredentialOptions

4.5. Additional options for Credential Generation (dictionary

#enumdef-attachmentReferenced in:
#enumdef-attachmentReferenced in:
#

#dictdef-scopedcredentialparametersReferenced in:
#dictdef-scopedcredentialparametersReferenced in:
#dictdef-scopedcredentialparametersReferenced in:
#dictdef-scopedcredentialparametersReferenced in:

#dom-scopedcredentialoptions-timeoutReferenced in:
* 4.1.1. Create a new credential - makeCredential() method (2)

#dom-scopedcredentialoptions-rpidReferenced in:
* 4.1.1. Create a new credential - makeCredential() method (2) (3)

#dom-scopedcredentialoptions-excludelistReferenced in:
* 4.1.1. Create a new credential - makeCredential() method (2) (3)

#dom-scopedcredentialoptions-attachmentReferenced in:
* 4.1.1. Create a new credential - makeCredential() method (2)

#dom-scopedcredentialoptions-extensionsReferenced in:
* 4.1.1. Create a new credential - makeCredential() method

* 8.3. Extending request parameters

#enumdef-attachmentReferenced in:
* 4.5. Additional options for Credential Generation (dictionary ScopedCredentialOptions)
* 4.5.1. Credential Attachment enumeration (enum Attachment)
# 3. Terminology

#basic-attestationReferenced in:
* 5.3.5.1. Privacy

#self-attestationReferenced in:
* 3. Terminology (2) (3) (4)
* 5.3. Credential Attestation
* 5.5.2. Attestation Certificate and Attestation Certificate CA
  Compromise

#privacy-caReferenced in:
* 5.3.5.1. Privacy

#direct-anonymous-attestationReferenced in:
* 5.3.5.1. Privacy

#attestation-statement-format-identifierReferenced in:
* 5.3.2. Attestation Statement Formats
* 5.3.4. Generating an Attestation Object

#client-argumentReferenced in:

#client-argumentReferenced in:

#client-argumentReferenced in:

#client-argumentReferenced in:

#attestation-dataReferenced in:
* 5.1. Authenticator data (2) (3) (4) (5) (6) (7)
* 5.2.1. The authenticatorMakeCredential operation
* 5.2.2. The authenticatorGetAssertion operation
* 5.3. Credential Attestation
* 5.3.3. Attestation Types
* 6.1. Registering a new credential
* 7.2. Packed Attestation Statement Format
* 7.3. TPM Attestation Statement Format
* 7.4. Android Key Attestation Statement Format

#authenticator-data-for-the-attestationReferenced in:
* 7.2. Packed Attestation Statement Format
* 7.3. TPM Attestation Statement Format
* 7.4. Android Key Attestation Statement Format (2)
* 7.5. Android SafetyNet Attestation Statement Format
* 7.6. FIDO U2F Attestation Statement Format

#authenticator-data-claimed-to-have-been-used-for-the-attestationReferenced in:
* 7.2. Packed Attestation Statement Format
* 7.3. TPM Attestation Statement Format
* 7.4. Android Key Attestation Statement Format (2)
* 7.6. FIDO U2F Attestation Statement Format

#basic-attestationReferenced in:
* 5.3.5.1. Privacy

#self-attestationReferenced in:
* 3. Terminology (2) (3) (4)
* 5.3. Credential Attestation
* 5.3.5.2. Attestation Certificate and Attestation Certificate CA
  Compromise

#privacy-caReferenced in:
* 5.3.5.1. Privacy

#elliptic-curve-based-direct-anonymous-attestationReferenced in:
* 5.3.5.1. Privacy

#ecdaaReferenced in:
* 5.3.2. Attestation Statement Formats
* 5.3.3. Attestation Types
* 5.3.5.2. Attestation Certificate and Attestation Certificate CA
  Compromise
* 6.1. Registering a new credential
* 7.2. Packed Attestation Statement Format (2)
* 7.3. TPM Attestation Statement Format (2)

#attribution-statement-format-identifierReferenced in:
* 5.3.2. Attestation Statement Formats
* 5.3.4. Generating an Attestation Object

#identifier-of-the-ecdaa-issuer-public-keyReferenced in:
* 6.1. Registering a new credential
* 7.2. Packed Attestation Statement Format
* 7.3. TPM Attestation Statement Format (2)

#ecdaa-issuer-public-keyReferenced in:
* 5.3.2. Attestation Statement Formats
* 5.3.5.1. Privacy
* 6.1. Registering a new credential
* 7.2. Packed Attestation Statement Format (2) (3)

#registration-extensionReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential
  \{Create\}(options) method
* 8. WebAuthn Extensions (2) (3) (4) (5) (6)
* 8.6. Example Extension
8.3. Extending request parameters

#authentication-extensionReferenced in:
* 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]()[options)] method
* 8. WebAuthn Extensions (2) (3) (4) (5)

#extension-identifierReferenced in:
* 8.5. Authenticator extension processing
* 8.3. Extending request parameters
* 8.2. Defining extensions

#client-extensionReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's [[Create]()[options)] method
* 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]()[options)] method
* 4.7. Authentication Extensions (typedef AuthenticationExtensions)
* 8. WebAuthn Extensions
* 8.2. Defining extensions
* 8.4. Client extension processing

#client-extension-inputReferenced in:
* 8. WebAuthn Extensions
* 8.3. Extending request parameters
* 8.2. Defining extensions
* 8.1.4. Use an existing credential - PublicKeyCredential's [[Create]()[options)] method

#client-extension-inputReferenced in:
* 8. WebAuthn Extensions
* 8.3. Extending request parameters
* 8.2. Defining extensions
* 8.1.4. Use an existing credential - PublicKeyCredential's [[Create]()[options)] method

#authentication-extension-inputReferenced in:
* 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]()[options)] method
* 8. WebAuthn Extensions (2) (3) (4) (5)
### 9.3. Authenticator Selection Extension

- **9.3.** Authenticator Selection Extension

### 8. WebAuthn Extensions (2) (3) (4) (5)

- **8.2.** Defining extensions
- **8.3.** Extending request parameters (2) (3)
- **8.4.** Client extension processing
- **8.5.** Authenticator extension processing (2) (3)

### 8. WebAuthn Extensions (2) (3) (4)

- **8.2.** Defining extensions

- **8.2.** Client extension processing (2) (3)
- **8.4.** Client extension processing (2) (3)
- **8.6.** Example Extension

### 9.4. Authenticator Selection Extension (authnSel)

- **9.4.** Authenticator Selection Extension (authnSel)

### 9.4. Authenticator Selection Extension (authnSel)

- **9.4.** Authenticator Selection Extension (authnSel)