Abstract

Copyright 2017 W3C^ (MIT, ERCIM, Keio, Beihang), W3C liability, trademark and document use rules apply.

Tests:
- web-platform-tests webauthn/ (ongoing work)

THE TITLE: Web Authentication: An API for accessing Public Key Credentials - Level 1

W3C

GitHub

Issue Tracking:
- https://www.w3.org/TR/2016/WD-webauthn-20160531/
- https://www.w3.org/TR/2016/WD-webauthn-20160928/
- https://www.w3.org/TR/2017/WD-webauthn-20170216/
- https://www.w3.org/TR/2017/WD-webauthn-20170811/
- https://www.w3.org/TR/2017/WD-webauthn-20170928/
- https://www.w3.org/TR/2018/WD-webauthn-20180315/

Previous versions:
- https://www.w3.org/TR/2016/WD-webauthn-20160531/
- https://www.w3.org/TR/2016/WD-webauthn-20160928/
- https://www.w3.org/TR/2017/WD-webauthn-20170216/
- https://www.w3.org/TR/2017/WD-webauthn-20170811/
- https://www.w3.org/TR/2017/WD-webauthn-20170928/
- https://www.w3.org/TR/2018/WD-webauthn-20180315/

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

Editors:
- Emil Lundberg (Yubico)
- Rolf Lindemann (Nok Nok Labs)
- Angelo Liao (Microsoft)
- Akshay Kumar (Microsoft)
- Michael B. Jones (Microsoft)
- Dirk Balfanz (Google)
- Jeff Hodges (PayPal)

Contributors:
- Chritiaan Brand (Google)
- Adam Langley (Google)
- Girishd Mandyam (Qualcomm)
- Mike West (Google)
- Jeffrey Yasskin (Google)

Jeffrey Yasskin (Google)
Giridhar Mandyam (Qualcomm)
Adam Langley (Google)

Editors:
- Emil Lundberg (Yubico)
- Rolf Lindemann (Nok Nok Labs)
- Angelo Liao (Microsoft)
- Akshay Kumar (Microsoft)
- Michael B. Jones (Microsoft)
- Dirk Balfanz (Google)
- Jeff Hodges (PayPal)

Contributors:
- Chritiaan Brand (Google)
- Adam Langley (Google)
- Girishd Mandyam (Qualcomm)
- Mike West (Google)
- Jeffrey Yasskin (Google)

Tests:
- web-platform-tests webauthn/ (ongoing work)

THE TITLE: Web Authentication: An API for accessing Public Key Credentials Level 1

W3C

Copyright 2018 W3C^ (MIT, ERCIM, Keio, Beihang), W3C liability, trademark and document use rules apply.

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 public key 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 https://www.w3.org/TR/.

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

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

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

This document is governed by the 1 March 2017 W3C Process Document.

Table of Contents
1. Introduction
   1.1 Use Cases
   1.1.1 Registration
   1.1.2 Authentication
   1.1.3 Other use cases and configurations
   1.2 Conformance
   1.2.1 User Agents

For the Web Authentication specification to move to Proposed Recommendation we must show two independent, interoperable implementations of the Web Authentication API. W3C will also have multiple interoperable implementations of the AppID extension, validating the extensions framework. All other extensions are "at risk". If there are not multiple interoperable implementations, each may independently be removed or made informative at Proposed Recommendation.

We have had two informal interoperability tests with implementations in three browsers. There is no preliminary implementation report at this time.

This document was published by the Web Authentication Working Group as a Candidate Recommendation. This document is intended to become a W3C Recommendation. Feedback and comments on this specification are welcome. Please use Github issues. Discussions may also be found in the public-webauthn@w3.org archives. W3C publishes a Candidate Recommendation to indicate that the document is believed to be stable and to encourage implementation by the developer community.

The deadline for comments for this Candidate Recommendation is 1 May 2018.

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

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

This document is governed by the 1 February 2018 W3C Process Document.

Table of Contents
1. Introduction
   1.1 Use Cases
   1.1.1 Registration
   1.1.2 Authentication
   1.1.3 Other use cases and configurations
   2. Conformance
   2.1 User Agents
2. 5.2 Authenticator Responses (interface AuthenticatorResponse)

1. 5.2.1 Information about Public Key Credential (interface AuthenticatorAttestationResponse)

2. 5.2.2 Web Authentication Assertion (interface AuthenticatorAssertionResponse)

3. 5.3 Parameters for Credential Generation (dictionary PublicKeyCredentialParameters)

3. 5.4 Options for Credential Creation (dictionary MakePublicKeyCredentialOptions)

1. 5.4.1 Public Key Entity Description (dictionary PublicKeyCredentialEntity)

2. 5.4.2 RP Parameters for Credential Generation (dictionary PublicKeyCredentialRpEntity)

3. 5.4.3 User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity)

4. 5.4.4 Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria)

5. 5.4.5 Authenticator Attachment enumeration (enum AuthenticatorAttachment)

6. 5.4.6 Attestation Conveyance Preference enumeration (enum AttestationConveyancePreference)

5. 5.5 Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)

6. 5.6 Abort operations with AbortSignal

7. 5.7 Authentication Extensions (typedef AuthenticationExtensions)

8. 5.8 Supporting Data Structures

1. 5.8.1 Client data used in WebAuthn signatures (dictionary CollectedClientData)

2. 5.8.2 Credential Type enumeration (enum PublicKeyCredentialType)

3. 5.8.3 Credential Descriptor (dictionary PublicKeyCredentialDescriptor)

4. 5.8.4 Authenticator Transport enumeration (enum AuthenticatorTransport)

5. 5.8.5 Cryptographic Algorithm Identifier (typedef COSEAlgorithmIdentifier)

6. 5.8.6 User Verification Requirement enumeration (enum COSEAlgorithmIdentifier)
1. 13.1 Cryptographic Challenges

2. 12.2 Registration Specifically with User Verifying Platform

3. 11.3 COSE Algorithm Registrations

4. 10.4 Authenticator Selection Extension (authnSel)

5. 10.5 Supported Extensions Extension (exts)

6. 10.6 User Verification Index Extension (uvi)

7. 10.7 Location Extension (loc)

8. 10.8 User Verification Method Extension (uvm)

9. 10.9 Biometric Authenticator Performance Bounds Extension (biometricPerfBounds)

10. 10 Defined Extensions

11. 11 IANA Considerations

12. 12 Sample scenarios

13. 13 Security Considerations

14. 14 Acknowledgements

15. Index
1.1. Use Cases

This section is not normative.

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. A public key credential is created and stored by an authenticator at the behest of a Relying Party, subject to user consent. Subsequently, the public key 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. Other 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) integrated into the computing device in conjunction with some means for user verification, along with appropriate Compromiseaware middleware 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 12 Sample scenarios.

1.1.1. Registration

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 12 Sample scenarios.
+ 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.”

2.1. User Agents

A User Agent MUST behave as described by 5 Web Authentication API in order to be considered conformant. Conforming User Agents MAY implement algorithms given in this specification in any way desired, so long as the result is indistinguishable from the result that would be obtained by the specification’s algorithms.

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

2.2. Authenticators

An authenticator MUST provide the operations defined by 6 WebAuthn Authenticator model, and those operations MUST behave as described there. This is a set of functional and security requirements for an authenticator to be usable by a Conforming User Agent.

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 a 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 phone.

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.
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.

### 2.3. Relying Parties

A Relying Party MUST behave as described in 7 Relying Party Operations to get the security benefits offered by this specification.

### 3. Dependencies

This specification relies on several other underlying specifications, listed below and in Terms defined by reference.

#### 3.1. Base64url encoding

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

#### 3.2. CBOR

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

#### 3.3. CDDL

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

#### 3.4. COSE

COSE Object Signing and Encryption (COSE) [RFC8152]. The IANA COSE Algorithms registry established by this specification is also used.

#### 3.5. Credential Management

The API described in this document is an extension of the Credential concept defined in [CREDENTIAL-MANAGEMENT-1].

#### 3.6. DOM

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

#### 3.7. ECMAScript

%ArrayBuffer% is defined in [ECMAScript].

#### 3.8. HTML

The concepts of relevant settings object, origin, opaque origin, and is a registrable domain suffix of or is equal to are defined in [HTML5].

#### 3.9. Web IDL

Many of the interface definitions and all of the IDL in this specification depend on [WebIDL-1]. This updated version of the
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].

4. Terminology

4.1. Authentication Assertion

Authentication Assertion, a challenge and other data presented then cryptographically signing and returning, in the form of an and (ii) authenticate by potentially verifying the user, and a public key credential and register it with a Relying Party, A cryptographic entity used by a WebAuthn Client to (i) generate

This corresponds to the [CREDENTIAL-MANAGEMENT-1] specification’s single-use credentials.

Authenticator

A cryptographic entity used by a WebAuthn Client to (i) generate a public key credential and register it with a Relying Party, and (ii) authenticate by potentially verifying the user, and then cryptographically signing and returning, in the form of an

Authorization Gesture

The ceremony where a user, and the user's computing device(s) work in concert to confirm, or authenticate. In the WebAuthn context, attestation is employed to enhance an authenticator and the data it emits; including, for example: credential IDs, credential key pairs, signature counters, etc. An attestation statement is conveyed in an attestation object during registration. See also 6.3 Attestation and Figure 3. Whether or how the client platform conveys the attestation statement and AAGUID portions of the attestation object to the Relying Party is described by attestation conveyance.

The ceremony where a user, and the user's computing device(s) (containing at least one authenticator), 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 credential private key associated with a previously-registered public key credential (see Registration). Note that this includes a test of user presence or user verification. This corresponds to the [CREDENTIAL-MANAGEMENT-1] specification's single-use credentials. 

4.1.2. See Authentication Assertion.

4.2. Attestation

Generally, attestation is a statement serving to bear witness, confirm, or authenticate. In the WebAuthn context, attestation is employed to enhance an authenticator and the data it emits; including, for example: credential IDs, credential key pairs, signature counters, etc. An attestation statement is conveyed in an attestation object during registration. See also 6.3 Attestation and Figure 3. Whether or how the client platform conveys the attestation statement and AAGUID portions of the attestation object to the Relying Party is described by attestation conveyance.

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 nor attestation certificate, see self attestation for details.

4.3.3. See Authentication Assertion.

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

4.5. Authentication Assertion

The cryptographically signed AuthenticatorAssertionResponse object returned by an authenticator as the result of a authenticatorMakeCredential operation.

This corresponds to the [CREDENTIAL-MANAGEMENT-1] specification's single-use credentials.

4.6. Authenticator

A cryptographic entity used by a WebAuthn Client to (i) generate a public key credential and register it with a Relying Party, and (ii) authenticate by potentially verifying the user, and then cryptographically signing and returning, in the form of an Authentication Assertion, a challenge and other data presented by a Relying Party (in concert with the WebAuthn Client).

4.4. Authorization Gesture

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].
An authorization gesture is a physical interaction performed by
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
case 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, this
Registration and Authentication are ceremonies, and an
authorization gesture is often a component of those ceremonies.

Client
See Conforming User Agent.

Client-Side
This refers in general to the combination of the user’s platform
device, user agent, authenticators, and everything gluing it all
together.

Client-side-resident Credential Private Key
A Client-side-resident Credential Private Key is stored either
on the client platform, or in some case on the authenticator
itself, e.g., in the case of a discrete first-factor roaming
authenticator. Such client-side credential private key storage
has the property that the authenticator is able to select the
credential private key given only an RP ID, possibly with user
assistance (e.g., by providing the user a pick list of credentials
associated with the RP ID). By definition, the
private key is always exclusively controlled by the
Authenticator. In the case of a Client-side-resident Credential
Private Key, the Authenticator might offload storage of wrapped
key material to the client platform, but the client platform is
not expected to offload the key storage to remote entities (e.g.
RP Server).

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.

Credentila ID
A probabilistically-unique byte sequence identifying a public
key credential source and its authentication assertions.

Credentila IDs are generated by authenticators in two forms:

1. At least 16 bytes that include at least 100 bits of entropy,
or
2. The public key credential source, without its Credential ID,
   encrypted so only its managing authenticator can decrypt it.
   This form allows the authenticator to be nearly stateless, by
   having the Relying Party store any necessary state.

Note: [FIDO-UAF-AUTHNR-CMDS] includes guidance on encryption
techniques under "Security Guidelines".
Relying Parties do not need to distinguish these two Credential ID forms.

Credible 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.

Public Key Credential Source

A credential source ([CREDENTIAL-MANAGEMENT-1]) used by an authenticator to generate authentication assertions. A public key credential source has:

- A Credential ID.
- A credential private key.
- The Relying Party Identifier for the Relying Party that created this credential source.
- An optional user handle for the person who created this credential source.
- Optional other information used by the authenticator to inform its UI. For example, this might include the user’s display name.

The authenticatorMakeCredential operation creates a public key credential source bound to a managing authenticator and returns the credential public key associated with its credential private key. The Relying Party can use this credential public key to verify the authentication assertions created by this public key credential source.

Public Key Credential

Generically, a credential is data one entity presents to another in order to authenticate the former to the latter ([RFC4949]). The term public key credential refers to one of: a public key credential source, the possibly-attested credential public key corresponding to a public key credential source, or an authentication assertion. Which one is generally determined by context.

Note: This is a willful violation of [RFC4949]. In English, a "credential" is both a) the thing presented to prove a statement and b) intended to be used multiple times. It's impossible to achieve both criteria securely with a single piece of data in a public key system. [RFC4949] chooses to define a credential as the thing that can be used multiple times (the public key),
while this specification gives "credential" the English term’s flexibility. This specification uses more specific terms to identify the data related to an [RFC4949] credential:

“Authentication information” (possibly including a private key)
Public key credential source

“Signed value”

Authentication assertion

[RFC4949] "credential"
Credentila public key or attestation object

At registration time, the authenticator creates an asymmetric key pair, and stores its private key portion and information from the Relying Party into a public key credential source. The public key portion is returned to the Relying Party, who then stores it in conjunction with the present user’s account. Subsequently, only that Relying Party, as identified by its RP ID, is able to employ the public key credential for authentication ceremonies, via the get() method. The Relying Party uses its stored copy of the credential public key to verify the resultant authentication assertion.

Rate Limiting

The process (also known as throttling) by which an authenticator implements controls against brute force attacks by limiting the number of consecutive failed authentication attempts within a given period of time. If the limit is reached, the authenticator should impose a delay that increases exponentially with each successive attempt, or disable the current authentication modality and offer a different authentication factor if available. Rate limiting is often implemented as an aspect of user verification.

Registration

The ceremony where a user, a Relying Party, and the user’s computing device(s) (containing at least one authenticator) work in concert to create a public key credential and associate it with the user’s Relying Party account. Note that this includes employing a test of user presence or 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

A valid domain string that identifies the Relying Party on whose behalf a given registration or authentication ceremony is being performed. A public key credential can only be used for authentication with the same entity (as identified by RP ID) it was registered with. By default, the RP ID for a WebAuthn operation is set to the caller’s origin’s effective domain. See also 5.1.3 Create a new credential - PublicKeyCredential’s [[Create][origin, options, sameOriginWithAncestors]] method and 5.1.4 Use an existing credential to make an assertion - PublicKeyCredential’s [[Get][options]] method.

Note: A Public key credential’s scope is for a Relying Party’s origin, with the following restrictions and relaxations:
User Agent. A WebAuthn Client is an intermediary entity that interfaces 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 a user presence test, by definition, is not capable of biometric recognition, nor does it involve the presentation of a shared secret such as a password or PIN.

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.

User Handle
The user handle is specified by a Relying Party and is a unique identifier for a user account with that Relying Party. A user handle is an opaque byte sequence with a maximum size of 64 bytes. The user handle is not meant to be displayed to the user, but 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 user handle.

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). The intent is to be able to distinguish individual users. Note that invocation of the authenticatorMakeCredential and authenticatorGetAssertion operations implies use of key material managed by the authenticator. Note that for security, user verification and use of credential private keys must occur within a single logical security boundary defining the authenticator.

User Present
Upon successful completion of a user presence test, the user is said to be "present".

User Verified
Upon successful completion of a user verification process, the user is said to be "verified".

WebAuthn Client
Also referred to herein as simply a client. See also Conforming User Agent.
5. 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 Relying Party 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 and are mediated by the browser and/or platform on the user's behalf. At no point does the script get access to the credentials themselves; it only gets information about the credentials in the form of objects.

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. Authenticator 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 the credentials, ensures that all operations are scoped to a particular origin, and cannot be replayed against a different origin, to protect the origin in its responses. Specifically, as defined in 6.2 Authentication 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 the presence of public key 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 the Relying Party's origin and RP ID to the authenticator. Since this is an integral part of the WebAuthn security model, user agents must choose 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.

5.1. PublicKeyCredential Interface

The PublicKeyCredential interface inherits from Credential

[SECURE_CONTEXT, EXPOSED=WINDOW], and contains the attributes that are returned to the caller when a new credential is created, or a new assertion is requested.

5.2. PublicKeyCredential.1
interface PublicKeyCredential : Credential {
    [SameObject] readonly attribute ArrayBuffer rawId;
    [SameObject] readonly attribute AuthenticatorResponse response;
    [SameObject] AuthenticationExtensions getClientExtensionResults();
};

interface AuthenticationExtensions {
    [SameObject] AuthenticationExtensionsClientOutputs getClientExtensionResults();
};

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

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

response, of type AuthenticatorResponse, readonly
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
ever be an AuthenticatorAttestationResponse, otherwise, the
PublicKeyCredential was created in response to get(), and this
attribute’s value will be an AuthenticatorAssertionResponse.

getClientExtensionResults()
This operation returns the value of [[clientExtensionsResults]],
which is a map containing extension identifier -> client
extension output entries produced by the extension’s client
test extension processing.

[[type]]
The PublicKeyCredential interface object’s [[type]] internal
slot’s value is the string “public-key”.

Note: This is reflected via the type attribute getter inherited
from Credential.

[[discovery]]
The PublicKeyCredential interface object’s [[discovery]]
internal slot’s value is “remote”.

[[identifier]]
This internal slot contains an identifier for the credential,
chosen by the platform with help from the authenticator. This
identifier is used to look up credentials on the platform through
on-board storage may create identifiers containing a
private key wrapped with a symmetric key that is burned into the authenticator.

[[clientExtensionsResults]]
This internal slot contains the results of processing client
extensions requested by the Relying Party on the Relying
Party’s invocation of either navigator.credentials.create() or
navigator.credentials.get().

PublicKeyCredential’s interface object inherits Credential’s
implementation of [[CollectFromCredentialStore]]([origin, options],
sameOriginWithAncestors), and defines its own implementation of
[[Create]]([origin, options], sameOriginWithAncestors),
[[DiscoverFromExternalSource]]([origin, options],
sameOriginWithAncestors), and [[Store]]([credential],
sameOriginWithAncestors).
5.1. CredentialCreationOptions Extension

To support registration via `navigator.credentials.create()`, this document extends the CredentialCreationOptions dictionary as follows:

```
partial dictionary CredentialCreationOptions {
  publicKey: PublicKeyCredentialCreationOptions publicKey;
};
```

5.1.2. CredentialRequestOptions Extension

To support obtaining assertions via `navigator.credentials.get()`, this document extends the CredentialRequestOptions dictionary as follows:

```
partial dictionary CredentialRequestOptions {
  publicKey: PublicKeyCredentialRequestOptions publicKey;
};
```

5.1.3. Create a new credential - `PublicKeyCredential`'s `[[Create]]` method

`PublicKeyCredential`'s interface object’s implementation of the `[[Create]]` method:

```
[[Create]]((origin, options, sameOriginWithAncestors) internal method)
```

This internal method accepts three arguments:

- `origin`: This argument is the relevant settings object’s origin, as determined by the calling create() implementation.
- `options`: This argument is a CredentialCreationOptions object whose options.publicKey member contains a PublicKeyCredentialCreationOptions object specifying the desired attributes of the to-be-created public key credential.
- `sameOriginWithAncestors`: This argument is a boolean which is true if and only if the caller’s environment settings object is same-origin with its ancestors.

Note: This algorithm is synchronous: the Promise resolution/rejection is handled by `navigator.credentials.create()`.

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

1. Assert: options.publicKey is present.
2. If sameOriginWithAncestors is false, return a "NotAllowedError" DOMException.

Note: This “sameOriginWithAncestors” restriction aims to address the concern raised in the Origin Conflation section of [CREDENTIAL-MANAGEMENT-1], while allowing Relying Party script access to Web Authentication functionality, e.g., when running in a secure context framed document that is same-origin with its ancestors. However, in the future, this specification (in conjunction with [CREDENTIAL-MANAGEMENT-1]) may provide Relying Parties with more fine-grained control—e.g., range from allowing only top-level access to Web Authentication functionality, to allowing cross-origin embedded cases—by leveraging [Feature-Policy] once the latter specification becomes stably implemented in user agents.
3. L et options be the value of options.publicKey.
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 a

5.1.1. CredentialCreationOptions Dictionary Extension

To support registration via `navigator.credentials.create()`, this document extends the CredentialCreationOptions dictionary as follows:

```
partial dictionary CredentialCreationOptions {
  PublicKeyCredentialCreationOptions publicKey;
};
```

5.1.2. CredentialRequestOptions Dictionary Extension

To support obtaining assertions via `navigator.credentials.get()`, this document extends the CredentialRequestOptions dictionary as follows:

```
partial dictionary CredentialRequestOptions {
  PublicKeyCredentialRequestOptions publicKey;
};
```

5.1.3. Create a new credential - `PublicKeyCredential`'s `[[Create]]` method

`PublicKeyCredential`'s interface object’s implementation of the `[[Create]]` method:

```
[[Create]]((origin, options, sameOriginWithAncestors) internal method)
```

This internal method accepts three arguments:

- `origin`: This argument is the relevant settings object’s origin, as determined by the calling create() implementation.
- `options`: This argument is a CredentialCreationOptions object whose options.publicKey member contains a PublicKeyCredentialCreationOptions object specifying the desired attributes of the to-be-created public key credential.
- `sameOriginWithAncestors`: This argument is a boolean which is true if and only if the caller’s environment settings object is same-origin with its ancestors.

Note: This algorithm is synchronous: the Promise resolution/rejection is handled by `navigator.credentials.create()`.

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

1. Assert: options.publicKey is present.
2. If sameOriginWithAncestors is false, return a "NotAllowedError" DOMException.

Note: This “sameOriginWithAncestors” restriction aims to address the concern raised in the Origin Conflation section of [CREDENTIAL-MANAGEMENT-1], while allowing Relying Party script access to Web Authentication functionality, e.g., when running in a secure context framed document that is same-origin with its ancestors. However, in the future, this specification (in conjunction with [CREDENTIAL-MANAGEMENT-1]) may provide Relying Parties with more fine-grained control—e.g., range from allowing only top-level access to Web Authentication functionality, to allowing cross-origin embedded cases—by leveraging [Feature-Policy] once the latter specification becomes stably implemented in user agents.
3. L et options be the value of options.publicKey.
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 a
1. Let callerOrigin be origin. If callerOrigin is an opaque origin, return a DOMException whose name is "NotAllowedError", and terminate this algorithm.
2. Set clientExtensions[extensionId] to clientExtensionInput. If the algorithm returned an error, running extensionId's client extension processing algorithm on clientExtensionInput.
3. Append the pair of current.type and alg to credTypesAndPubKeyAlgs.
4. Append the pair of current.type and alg to credTypesAndPubKeyAlgs.
5. Set authenticatorExtensions[extensionId] to the base64url encoding of authenticatorExtensionInput.
6. Let collectedClientData be a new CollectedClientData instance whose fields are:
   - type: The string "webauthn.create".
   - challenge: The base64url encoding of options.challenge.
   - origin: The serialization of callerOrigin.
   - hashAlgorithm: The recognized algorithm name of the hash algorithm selected by the client for generating the hash of the serialized client data.
7. If options.id is present
   - If options.id is not a registrable domain suffix of and is not equal to effectiveDomain, return a DOMException whose name is "SecurityError", and terminate this algorithm.
   - Set options.id to effectiveDomain.
   - Note: options.rp.id represents the caller's RP ID. The RP ID defaults to being the caller's origin's effective domain unless the caller has explicitly set options.rp.id when calling create().
   - Let credTypesAndPubKeyAlgs be a new list whose items are pairs of PublicKeyCredentialType and a COSEAlgorithmIdentifier.
   - 3. Append the pair of current.type and alg to credTypesAndPubKeyAlgs.
   - 5. Let extensionId be extensionId. If extensionId is not a registration extension, then continue.
   - 8. Let collectedClientData be a new CollectedClientData instance whose fields are:
   - type: The string "webauthn.create".
   - challenge: The base64url encoding of options.challenge.
   - origin: The serialization of callerOrigin.
   - hashAlgorithm: The recognized algorithm name of the hash algorithm selected by the client for generating the hash of the serialized client data.
   - timer timerLifetimeTimer to this adjusted value, if the timeout member of options is not present, then set timerLifetimeTimer to a platform-specific default.
   - 5. Let callerOrigin be origin. If callerOrigin is an opaque origin, return a DOMException whose name is "NotAllowedError", and terminate this algorithm.
   - 6. Let effectiveDomain be the callerOrigin's effective domain. If effective domain is not a valid domain, then return a DOMException whose name is "SecurityError" and terminate this algorithm.
   - Note: An effective domain may resolve to a host, which can be represented in various manners, such as domain name, ipv4 address, ipv6 address, opaque host, or empty host. Only the domain format of host allowed here.
7. If options.id is not present
   - Append the pair of current.type and alg to credTypesAndPubKeyAlgs.
   - 3. Append the pair of current.type and alg to credTypesAndPubKeyAlgs.
   - 1. If current.type does not contain a PublicKeyCredentialType supported by this implementation, then continue.
   - 2. Let alg be current.alg.
   - 3. Let collectedClientData be a new CollectedClientData instance whose fields are:
   - type: The string "webauthn.create".
   - challenge: The base64url encoding of options.challenge.
   - origin: The serialization of callerOrigin.
   - hashAlgorithm: The recognized algorithm name of the hash algorithm selected by the client for generating the hash of the serialized client data.
   - 5. Let extensionId be extensionId. If extensionId is not a registration extension, then continue.
   - 8. Let collectedClientData be a new CollectedClientData instance whose fields are:
   - type: The string "webauthn.create".
   - challenge: The base64url encoding of options.challenge.
   - origin: The serialization of callerOrigin.
14. Let clientDataJSON be the JSON-serialized client data constructed from collectedClientData.

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

16. If the options.signal is present and its aborted flag is set to true, return a DOMException whose name is "AbortError" and terminate this algorithm.

17. Start lifetimeTimer.

18. Let issuedRequests be a new ordered set.

19. For each authenticator that becomes available on this platform during the lifetime of lifetimeTimer, do the following:

20. If options.authenticatorSelection is present:

21. If options.authenticatorSelection.authenticatorAttachment is set to preferred, continue.

22. Let userVerification be the effective user verification requirement for credential creation, a Boolean value, as follows. If options.authenticatorSelection.userVerification is set to required and the authenticator is not capable of performing user verification, continue.

23. Let userPresence be the Boolean value set to the inverse of userVerification.

24. Let userVerification be true.

25. If the authenticator is capable of user verification:

26. Let userVerification be true.

27. If the authenticator is not capable of user verification:

28. Let userVerification be false.

29. If options.authenticatorSelectionexcludeCredentialDescriptorList, and authenticatorExtensions is present:

30. If options.authenticatorSelection.authenticatorAttachment is not equal to authenticator's attachment modality, continue.

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

32. If options.authenticatorSelection.userVerification is set to required and the authenticator is not capable of performing user verification, continue.

33. Let excludeCredentialDescriptorList be a new list.

34. If options.excludeCredentials:

35. Otherwise, Append C to excludeCredentialDescriptorList.

36. Let userPresence be a Boolean value set to the inverse of userVerification.

37. For each credential descriptor C in options.excludeCredentials:

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

39. Otherwise, Append C to excludeCredentialDescriptorList.

40. Let clientExtensions be the JSON-serialized client data constructed from collectedClientData.

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

42. If the options.signal is present and its aborted flag is set to true, return a DOMException whose name is "AbortError" and terminate this algorithm.

43. Start lifetimeTimer.

44. Let issuedRequests be a new ordered set.

45. For each authenticator that becomes available on this platform during the lifetime of lifetimeTimer, do the following:

46. If options.authenticatorSelection is present:

47. If options.authenticatorSelection.authenticatorAttachment is set to preferred, continue.

48. Let userVerification be the effective user verification requirement for credential creation, a Boolean value, as follows. If options.authenticatorSelection.userVerification is set to required and the authenticator is not capable of performing user verification, continue.

49. Let userPresence be the Boolean value set to the inverse of userVerification.

50. If the authenticator is capable of user verification:

51. Let userVerification be true.

52. If the authenticator is not capable of user verification:

53. Let userVerification be false.

54. If userVerification is set to discouraged:

55. Let userVerification be false.

56. For each credential descriptor C in options.excludeCredentials:

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

58. Otherwise, Append C to excludeCredentialDescriptorList.

59. Invoke the authenticatorMakeCredential operation on authenticator with clientDataHash, options, user, options.authenticatorSelection.requireResidentKey, userPresence, userVerification, credTypesAndPubKeyAlgs, excludeCredentialDescriptorList, and authenticatorExtensions: 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 excludeCredentialDescriptorList.

3. Invoke the authenticatorMakeCredential operation on authenticator with clientDataHash, options, user, options.authenticatorSelection.requireResidentKey, userPresence, userVerification, credTypesAndPubKeyAlgs, excludeCredentialDescriptorList, and authenticatorExtensions:
6.3.4 Generating an Attestation Object.

Note: this value is attObj, as defined in

whose value is the bytes returned from the

2. Let credentialCreationData be a struct whose items are:

1. Remove authenticator from issuedRequests.

If any authenticator indicates success,

If any authenticator returns a status indicating that the user

cancelled the operation,

If any authenticator returns an error status;

Remove authenticator from issuedRequests.

If any authenticator indicates success,

If any authenticator returns a status equivalent to

"InvalidStateError",

1. Remove authenticator from issuedRequests.

2. Let credentialCreationData be a struct whose items are:

2. Removing authenticator from issuedRequests:

For each authenticator in issuedRequests invoke the

authenticatorCancel operation on authenticator and remove

authenticator from issuedRequests. Then return a

DOMException whose name is "AbortError" and terminate this

algorithm.

If any authenticator returns a status indicating that the user

cancelled the operation,

If the options.signal is present and its aborted flag is set to

true, For each authenticator in issuedRequests invoke the

authenticatorCancel operation on authenticator and remove

authenticator from issuedRequests. Then return a

DOMException whose name is "AbortError" and terminate this

algorithm.

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.

For each authenticator in issuedRequests invoke the

authenticatorCancel operation on authenticator and remove

it from issuedRequests.

If the options.signal is present and its aborted flag is set to

true, For each authenticator in issuedRequests invoke the

authenticatorCancel operation on authenticator and remove

authenticator from issuedRequests. Then return a

DOMException whose name is "AbortError" and terminate this

algorithm.

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.

For each authenticator in issuedRequests invoke the

authenticatorCancel operation on authenticator and remove

it from issuedRequests.

If the options.signal is present and its aborted flag is set to

true, For each authenticator in issuedRequests invoke the

authenticatorCancel operation on authenticator and remove

authenticator from issuedRequests. Then return a

DOMException whose name is "AbortError" and terminate this

algorithm.

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.

For each authenticator in issuedRequests invoke the

authenticatorCancel operation on authenticator and remove

it from issuedRequests.

If the options.signal is present and its aborted flag is set to

true, For each authenticator in issuedRequests invoke the

authenticatorCancel operation on authenticator and remove

authenticator from issuedRequests. Then return a

DOMException whose name is "AbortError" and terminate this

algorithm.

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.

For each authenticator in issuedRequests invoke the

authenticatorCancel operation on authenticator and remove

it from issuedRequests.
@balfanz wishes to add to the "direct" RP. unaltered, to the attest statement, by employing a Privacy CA).
example, by employing a Privacy CA).
example, by employing a Privacy CA).
verifiable version of the same data (for privacy-friendly and/or more easily

The client MAY replace the AAGUID and

credential creation data with 16 zero bytes.

1. If the AAGUID in the attested credential
data is 16 zero bytes, credentialCreationData.attestationObjectResult.attStmt to be an empty CBOR
dict, fmt is "packed", and "x5c" & "ecdaaKeyId" are both absent from the attestation object. Then self attestation is being used and no further action is needed.

2. Otherwise

1. Replace the AAGUID in the attested
credential data with 16 zero bytes.
2. Set the value of credentialCreationData.attestationObjectResult.fmt to "none", and set the value of credentialCreationData.attestationOptions to the value of attestationConveyancePreferenceOption.

Generating an Attestation Object).

"indirect" The client MAY replace the AAGUID and attest statement with a more privacy-friendly and/or more easily verifiable version of the same data (for example, by employing a Privacy CA).

"direct" Convey the authenticator's AAGUID and attest statement, unaltered, to the RP.

@balfanz wishes to add to the "direct"
uses 6.2.2 The authenticatorGetAssertion operation to sign a Relying Party script optionally specifies some criteria to indicate what credential sources are acceptable to it. The user agent and/or platform locates credential sources matching the specified criteria, and guides the user to pick one that the script will be allowed to use. The user may choose to decline the entire interaction even if a credential source is present, for example to maintain privacy. If the user picks a credential source, the user agent then

3. Let id be `attestationObject.authData.attestedCredentialData.clientId`.

4. Let pubKeyCred be a new PublicKeyCredential object associated with global whose fields are:

```
[[identifier]],
id, response
```

A New AuthenticatorAttestationResponse object associated with global whose fields are:

```
[[clientDataJSON]],
clientDataJSON
```

A new ArrayBuffer, created using global's `%ArrayBuffer%`, containing the bytes of credentialCreationData.clientData.

```
attestationObject,
```

`attestationObject`.

```
[[clientExtensionsResults]],
```

A new ArrayBuffer, created using global's `%ArrayBuffer%`, containing the bytes of credentialCreationData.clientExtensionResults.

```
suits.
```

5. Return `pubKeyCred`.

4. For each remaining authenticator in `issuedRequests` invoke

```
for each remaining authenticator in issuedRequests invoke
```

the authenticatorCancel operation on authenticator and remove it from `issuedRequests`.

5. Return `constructCredentialAlg` and terminate this algorithm.

21. 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.

5.1.4. Use an existing credential to make an assertion - PublicKeyCredential's `[[Get]](options)` method

Relying Parties call `navigator.credentials.get([publicKey:..., ...])` to discover and use an existing public key credential, with the user's consent. Relying Party script optionally specifies some criteria to indicate what credential sources are acceptable to it. The user agent and/or platform locates credential sources matching the specified criteria, and guides the user to pick one that the script will be allowed to use. The user may choose to decline the entire interaction even if a credential source is present, for example to maintain privacy. If the user picks a credential source, the user agent then uses 6.2.3 The authenticatorGetAssertion operation to sign a Relying

2. Let `attestationObject` be a new ArrayBuffer, created using global's `%ArrayBuffer%`, containing the bytes of credentialCreationData.attestationObjectResult's value.

3. Let id be `attestationObject.authData.attestedCredentialData.clientId`.

4. Let pubKeyCred be a new PublicKeyCredential object associated with global whose fields are:

```
[[identifier]],
id, response
```

A New AuthenticatorAttestationResponse object associated with global whose fields are:

```
[[clientDataJSON]],
clientDataJSON
```

A new ArrayBuffer, created using global's `%ArrayBuffer%`, containing the bytes of credentialCreationData.clientData.

```
attestationObject,
```

`attestationObject`.

```
[[clientExtensionsResults]],
```

A new ArrayBuffer, created using global's `%ArrayBuffer%`, containing the bytes of credentialCreationData.clientExtensionResults.

```
suits.
```

5. Return `pubKeyCred`.

4. For each remaining authenticator in `issuedRequests` invoke

```
for each remaining authenticator in issuedRequests invoke
```

the authenticatorCancel operation on authenticator and remove it from `issuedRequests`.

5. Return `constructCredentialAlg` and terminate this algorithm.

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

In order to prevent information leak that could identify the user without consent, this step MUST NOT be executed before lifetimeTimer has expired. See 14.3 Authentication Ceremony Privacy for details.
Party-provided challenge and other collected data into an assertion, which is used as a credential.

The (get) implementation [CREDENTIAL-MANAGEMENT-1] calls PublicKeyCredential([CollectFromCredentialStore]) to collect any credentials that should be available without user mediation (roughly, this specification's authorization gesture), and if it does not find exactly one of those, it then calls PublicKeyCredential([DiscoverFromExternalSource]) to have the user select a credential source.

Since this specification requires an authorization gesture to create any credentials, the PublicKeyCredential([CollectFromCredentialStore]) or, options, sameOriginWithAncestors) internal method inherits the default behavior of PublicKeyCredential([CollectFromCredentialStore]), of returning an empty set.

5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method

This internal method accepts three arguments:

- origin
  - This argument is the relevant settings object's origin, as determined by the calling get() implementation, i.e., CredentialsContainer's Request a Credential abstract operation.
- options
  - This argument is a CredentialRequestOptions object whose PublicKeyCredentialRequestOptions.object specifying the desired attributes of the public key credential to discover.
- sameOriginWithAncestors
  - This argument is a boolean which is true if and only if the caller's environment settings object is same-origin with its ancestors.

Note: This algorithm is synchronous: the Promise resolution/rejection is handled by navigator.credentials.get().

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

1. Assert: options.publicKey is present.
   - If sameOriginWithAncestors is false, return a "NotAllowedError" DOMException.

Note: This “sameOriginWithAncestors” restriction aims to address the concern raised in the Origin Confusion section of [CREDENTIAL-MANAGEMENT-1], while allowing Relying Party script access to Web Authentication functionality, e.g., when running in a secure context framed document that is same-origin with its ancestors. However, in the future, this specification (in conjunction with [CREDENTIAL-MANAGEMENT-1] may provide Relying Parties with more fine-grained control—e.g., ranging from allowing top-level access to Web Authentication functionality, to allowing cross-origin embedded cases—by leveraging [Feature-Policy] once the latter specification becomes stably implemented in user agents.

2. Let options be the value of options.publicKey.
   - 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 a timer lifetimeTimer to this adjusted value. If the timeout member of options is not present, then set lifetimeTimer to a platform-specific default.

3. Let callerOrigin be origin. If callerOrigin is an opaque origin, return a DOMException whose name is "NotAllowedError", and terminate this algorithm.

4. Let effectiveDomain be the callerOrigin's effective domain. If
during the lifetime of lifetimeTimer, perform the following steps:

1. If options.rpid is not a registrable domain suffix of and is not equal to effectiveDomain, return a DOMException whose name is "SecurityError" and terminate this algorithm.
2. Set rpid to options.rpid.
3. If rpid is not the caller's RP ID, then return a DOMException whose name is "AbortError" and terminate this algorithm.
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. Let authenticatorExtensions be a new map and let authenticatorExtensions[extensionId] be an empty map.
6. If extensionId is not an authenticator extension, then continue.
7. Set authenticatorExtensions[extensionId] to the base64url encoding of authenticatorExtensionInput.
8. Let clientExtensions be a new map and let clientExtensions[extensionId] be an empty map.
9. 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 an authentication extension, then continue.
   2. Set clientExtensions[extensionId] to clientExtensionInput.
   3. If extensionId is not an authentication 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.
10. Let collectedClientData be a new CollectedClientData instance whose fields are:
   - type: The string "webauthn.get".
   - challenge: The base64url encoding of options.challenge.
   - origin: The serialization of callerOrigin.
   - hashAlgorithm: The recognized algorithm name of the hash algorithm selected by the client for generating the hash of the serialized client data.
   - tokenBindingId: The Token Binding ID associated with callerOrigin, if one is available.
   - clientExtensions: A new map.
   - authenticatorExtensions: A new map.
11. Let clientDataJSON be the JSON-serialized client data constructed from collectedClientData.
12. Let clientDataHash be the hash of the serialized client data represented by clientDataJSON.
13. If the options.signal is present and its abort flag is set to true, return a DOMException whose name is "AbortError" and terminate this algorithm.
14. Let issuedRequests be a new ordered set.
15. Let authenticator be a platform-specific handle whose value identifies an authenticator.
17. For each authenticator that becomes available on this platform during the lifetime of lifetimeTimer, perform the following steps:
transports (for this authenticator) in transports (for this authenticator) in transports (for this authenticator) in

1. For each credential descriptor C in allowCredentialDescriptorList, append each value, if any, of C.transports to distinctTransports.

Note: This will aggregate only distinct values of transports (for this authenticator) in allowCredentialDescriptorList[0].id’s value (see operation for more information).

4. Let distinctTransports be a new ordered set.

3. If allowCredentialDescriptorList is empty, continue.

2. Execute a platform-specific procedure to determine which, if any, public key credentials described by options.allowCredentials are bound to this authenticator, by matching with rpId, options.allowCredentials.id, and options.allowCredentials.type. Set allowCredentialDescriptorList to this filtered list.

6. If allowCredentialDescriptorList is not empty:

1. Let distinctTransports be a new ordered set.

2. If allowCredentialDescriptorList has exactly one value, let savedCredentialId be a new PublicKeyCredentialDescriptor.id and set its value to allowCredentialDescriptorList[0].id’s value (see operation for more information).

The foregoing step may be incorrect, in that we are attempting to create savedCredentialId here and use it later below, and we do not have a global in which to allocate a place for it. Perhaps this is good enough? addendum: @jcjones feels the above step is likely good enough.

1. For each credential descriptor C in allowCredentialDescriptorList, append each value, if any, of C.transports to distinctTransports.

The definitions of “lifetime of” and “becomes available” intended to represent how devices are plugged into (USB) or discovered by (NFC) browsers, and are under-specified. Resolving this with good definitions or some other means will be addressed by resolving Issue #613.

1. If options.userVerification is set to required and the authenticator is not capable of performing user verification, continue.

2. Let userVerification be the effective user verification requirement for assertion, a Boolean value, as follows. If options.userVerification is set to preferred:

Let userVerification be true.

is set to preferred

If the authenticator

is capable of user verification

Let userVerification be true.

is not capable of user verification

Let userVerification be false.

is set to discouraged

Let userVerification be false.

3. Let userPresence be a Boolean value set to the inverse of userVerification.

4. Let allowCredentialDescriptorList be a new list.

5. If options.allowCredentials is not empty, execute a platform-specific procedure to determine which, if any, public key credentials described by options.allowCredentials are bound to this authenticator, by matching with rpId, options.allowCredentials.id, and options.allowCredentials.type. Set allowCredentialDescriptorList to this filtered list.

6. If allowCredentialDescriptorList is not empty:

1. Let distinctTransports be a new ordered set.

2. If allowCredentialDescriptorList has exactly one value, let savedCredentialId be a new PublicKeyCredentialDescriptor.id and set its value to allowCredentialDescriptorList[0].id’s value (see operation for more information).

The foregoing step may be incorrect, in that we are attempting to create savedCredentialId here and use it later below, and we do not have a global in which to allocate a place for it. Perhaps this is good enough? addendum: @jcjones feels the above step is likely good enough.

1. For each credential descriptor C in allowCredentialDescriptorList, append each value, if any, of C.transports to distinctTransports.

Note: This will aggregate only distinct values of transports (for this authenticator) in...
distinctTransports due to the properties of ordered sets.

2. If distinctTransports is not empty,

The client selects one transport value from distinctTransports, possibly incorporating local configuration knowledge of the appropriate transport to use with authenticator in making its selection.

Then, using transport, invoke the authenticatorGetAssertion operation on authenticator, with rpId, clientDataHash, allowCredentialDescriptorList, userPresence, userVerification and authenticatorExtensions as parameters.

is empty

Using local configuration knowledge of the appropriate transport to use with authenticator, invoke the authenticatorGetAssertion operation on authenticator with rpId, clientDataHash, allowCredentialDescriptorList, userPresence, userVerification and clientExtensions as parameters.

Note: In this case, the Relying Party did not supply a list of acceptable credential descriptors. Thus, the authenticator is being asked to exercise any credential it may possess that is bound to the Relying Party, as identified by rpId.

7. Append authenticator to issuedRequests.

18. While issuedRequests is not empty, perform the following actions depending upon lifetimeTimer and responses from the authenticators:

If lifetimeTimer expires,

For each authenticator in issuedRequests invoke the authenticatorCancel operation on authenticator and remove authenticator from issuedRequests.

If the signal member is present and the aborted flag is set to true,

For each authenticator in issuedRequests invoke the authenticatorCancel operation on authenticator and remove authenticator from issuedRequests. Then return DOMException whose name is "AbortError" and terminate this algorithm.

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.

Note: Authenticators may return an indication of "the user cancelled the entire operation". How a user agent
1. If any authenticator returns an error status, remove authenticator from issuedRequests.
2. If any authenticator indicates success, remove authenticator from issuedRequests.
3. Let constructAssertionAlg be an algorithm that takes a struct whose fields are:

   - credentialIdResult
     - If savedCredentialId exists, set the value of credentialIdResult to be the bytes of savedCredentialId. Otherwise, set the value of credentialIdResult to be the bytes of the credential ID returned from the successful authenticatorGetAssertion operation, as defined in 6.2.3
   - clientDataJSON
     - The clientDataJSON client extension outputs, for each client extension processing algorithm to create the client extension output entries. The entries are created by running each extension's client extension output entries for the clientExtensionResults associated with global whose fields are:
     - clientDataJSONResult
       - whose value is the bytes of clientDataJSON.
     - authenticatorDataResult
       - whose value is the bytes of the authenticator data returned by the authenticator.
     - signatureResult
       - whose value is the bytes of the signature value returned by the authenticator.
     - userHandleResult
       - whose value is the bytes of the user handle returned by the authenticator.
     - clientExtensionResults
       - whose value is an AuthenticationExtensions clientExtensionResults object containing extension identifier -> client extension output entries. The entries are created by running each extension's client extension output entries for the clientExtensionResults associated with global whose fields are:
     - clientDataJSON
       - whose value is the bytes of clientDataJSON.
     - authenticatorData
       - whose value is the bytes of the authenticatorDataResult.
     - signature
       - whose value is the bytes of the signature.
     - userHandle
       - whose value is the bytes of the userHandleResult.

4. If any authenticator returns an error status, remove authenticator from issuedRequests.
5. If any authenticator indicates success, remove authenticator from issuedRequests.
6. Let constructAssertionAlg be an algorithm that takes a struct whose fields are:

   - credentialIdResult
     - If savedCredentialId exists, set the value of credentialIdResult to be the bytes of savedCredentialId. Otherwise, set the value of credentialIdResult to be the bytes of the credential ID returned from the successful authenticatorGetAssertion operation, as defined in 6.2.3
   - clientDataJSON
     - The clientDataJSON client extension outputs, for each client extension processing algorithm to create the client extension output entries. The entries are created by running each extension's client extension output entries for the clientExtensionResults associated with global whose fields are:
     - clientDataJSON
       - whose value is the bytes of clientDataJSON.
     - authenticatorData
       - whose value is the bytes of the authenticatorDataResult.
     - signature
       - whose value is the bytes of the signature.
     - userHandle
       - whose value is the bytes of the userHandleResult.

7. Let constructAssertionAlg be an algorithm that takes a struct whose fields are:

   - [identifier]
     - A new ArrayBuffer, created using global's %ArrayBuffer%, containing the bytes of assertionCreationData.credentialIdResult.
   - response
     - A new AuthenticatorAssertionResponse object associated with global whose fields are:
     - clientDataJSON
       - A new ArrayBuffer, created using global's %ArrayBuffer%, containing the bytes of assertionCreationData.clientDataJSON.
     - ONResult.

8. If any authenticator returns an error status, remove authenticator from issuedRequests.
9. If any authenticator indicates success, remove authenticator from issuedRequests.
10. Let constructAssertionAlg be an algorithm that takes a struct whose fields are:

     - [identifier]
       - A new ArrayBuffer, created using global's %ArrayBuffer%, containing the bytes of assertionCreationData.credentialIdResult.
     - response
       - A new AuthenticatorAssertionResponse object associated with global whose fields are:
       - clientDataJSON
         - A new ArrayBuffer, created using global's %ArrayBuffer%, containing the bytes of assertionCreationData.clientDataJSON.
When this method is invoked, the user agent MUST execute the following:

- The `[[Store]]`(credential, sameOriginWithAncestors) method is not supported for Web Authentication's PublicKeyCredential type, so it always returns an error.

  - Note: This algorithm is synchronous: the Promise resolution/rejection is handled by navigator.credentials.store().

- This internal method accepts two arguments:
  - credential
  - This argument is a PublicKeyCredential object.

- This argument is a boolean which is true if and only if the caller’s environment settings object is same-origin with its ancestors.

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

  1. 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.

  2. Return `pubKeyCred`.

  3. For each remaining authenticator in `issuedRequests` invoke the `authenticatorCancel` operation on authenticator and remove it from `issuedRequests`.

  4. Return `constructAssertionAlg` and terminate this algorithm.

  5. Return a `DOMException` whose name is "NotAllowedError". In order to prevent information leak that could identify the user without consent, this step MUST NOT be executed before `lifetimeTimer` has expired. See 14.3 Authentication Ceremony Privacy for details.

- 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.

- This internal method accepts two arguments:
  - credential
  - This argument is a PublicKeyCredential object.

- This argument is a boolean which is true if and only if the caller’s environment settings object is same-origin with its ancestors.

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

  1. The `[[Store]]`(credential, sameOriginWithAncestors) method is not supported for Web Authentication’s PublicKeyCredential type, so it always returns an error.

  2. Note: This algorithm is synchronous: the Promise resolution/rejection is handled by navigator.credentials.store().

  3. This internal method accepts two arguments:
     - credential
     - This argument is a PublicKeyCredential object.

     4. The `[[Store]]`(credential, sameOriginWithAncestors) method is not supported for Web Authentication’s PublicKeyCredential type, so it always returns an error.

     5. 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.

     6. This internal method accepts two arguments:
        - credential
        - This argument is a PublicKeyCredential object.

        7. When this method is invoked, the user agent MUST execute the following
algorithm: 1. Return a DOMException whose name is "NotSupportedError", and terminate this algorithm

5.1.6. Availability of User-Verifying Platform Authenticator -

PublicKeyCredential's isUserVerifyingPlatformAuthenticatorAvailable() method

Relying Parties use this method to determine whether they can create a new credential using a user-verifying platform authenticator. Upon invocation, the client employs a platform-specific procedure to discover available user-verifying platform authenticators. If successful, the client then assesses whether the user is willing to create a credential using one of the available user-verifying platform authenticators. This assessment may include various factors, such as:

- Whether the user is running in private or incognito mode.
- Whether the user has previously expressed an unwillingness to create a new credential for this Relying Party, either through configuration or by declining a user interface interaction.
- The user's explicitly stated intentions, determined through user interaction.

If this assessment is affirmative, the promise is resolved with the value of True. Otherwise, the promise is resolved with the value of False. Based on the result, the Relying Party can take further actions to guide the user to create a credential.

This method has no arguments and returns a boolean value.

If the promise will return False, the client SHOULD wait a fixed period of time from the invocation of the method before returning False. This is done so that callers can not distinguish between the case where the user was unwilling to create a credential using one of the available user-verifying platform authenticators and the case where no user-verifying platform authenticator exists. Trying to make these cases indistinguishable is done in an attempt not to provide additional information that could be used for fingerprinting. A timeout value on the order of 10 minutes is recommended: this is enough time for successful user interactions to be performed but short enough that the dangling promise will still be resolved in a reasonably timely fashion.

5.2. Authenticator Responses (interface AuthenticatorResponse)

Authenticators respond to Relying Party requests by returning an object derived from the AuthenticatorResponse interface: [SecureContext, Exposed=Window]

interface AuthenticatorResponse { [SameObject] readonly attribute ArrayBuffer clientDataJSON; };

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().

algorithm: 1. Return a DOMException whose name is "NotSupportedError", and terminate this algorithm

5.1.6. Preventing silent access to an existing credential -

PublicKeyCredential's [preventSilentAccess][credential, sameOriginWithAncestors] method

Calling the [preventSilentAccess][credential, sameOriginWithAncestors] method will have no effect on effectivators that require an authorization gesture, but setting that flag may potentially exclude effectivators that can operate without user intervention.

This internal method accepts no arguments.

5.1.7. Availability of Platform Authenticator -

PublicKeyCredential's isUserPlatformAuthenticatorAvailable() method

Relying Parties use this method to determine whether they can create a new credential using a user-verifying platform authenticator. Upon invocation, the client employs a platform-specific procedure to discover available platform authenticators. If successful, the client then assesses whether the user is willing to create a new credential using one of the available platform authenticators. This assessment may include various factors, such as:

- Whether the user is running in private or incognito mode.
- Whether the user has previously expressed an unwillingness to create a new credential for this Relying Party, either through configuration or by declining a platform-specific interaction.
- The user's explicitly stated intentions, determined through platform interaction.

If this assessment is affirmative, the promise is resolved with the value of True. Otherwise, the promise is resolved with the value of False. Based on the result, the Relying Party can take further actions to guide the user to create a credential.

This method has no arguments and returns a boolean value.

If the promise will return False, the client SHOULD wait a fixed period of time from the invocation of the method before returning False. This is done so that callers cannot distinguish between the case where the user was unwilling to create a credential using one of the available user-verifying platform authenticators and the case where no user-verifying platform authenticator exists. Trying to make these cases indistinguishable is done in an attempt not to provide additional information that could be used for fingerprinting. A timeout value on the order of 10 minutes is recommended: this is enough time for successful user interactions to be performed but short enough that the dangling promise will still be resolved in a reasonably timely fashion.

5.2. Authenticator Responses (interface AuthenticatorResponse)

Authenticators respond to Relying Party requests by returning an object derived from the AuthenticatorResponse interface: [SecureContext, Exposed=Window]

interface AuthenticatorResponse { [SameObject] readonly attribute ArrayBuffer clientDataJSON; };

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().
interface AuthenticatorAttestationResponse : AuthenticatorResponse {
    [SameObject] readonly attribute ArrayBuffer? userHandle;
    [SameObject] readonly attribute ArrayBuffer| signature;
    [SecureContext, Exposed=Window]
    interface AuthenticatorAttestationResponse : AuthenticatorResponse {
        [SameObject] readonly attribute ArrayBuffer| attestationObject;

    clientDataJSON
        This attribute, inherited from AuthenticatorResponse, contains
        the JSON serialized client data (see 6.3 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 the attestation
        statement 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 6.3
        Attestation, 5.3.4 Generating an Attestation Object, and Figure
        3.
    }

    5.2.2. 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, Exposed=Window]
    interface AuthenticatorAssertionResponse : AuthenticatorResponse {
        [SameObject] readonly attribute ArrayBuffer| authenticatorData;
        [SameObject] readonly attribute ArrayBuffer| userHandle;
    }

    clientDataJSON
        This attribute, inherited from AuthenticatorResponse, contains
        the JSON serialized client data (see 5.1 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 6.1 Authenticator data.
    signature, of type ArrayBuffer, readonly
        This attribute contains the raw signature returned from the
        authenticator. See 6.2.3 The AuthenticatorGetAssertion

    5.2.1. Information about Public Key Credential (interface
AuthenticatorAttestationResponse)

    The AuthenticatorAttestationResponse interface represents the
    authenticator’s response to a client’s request for generation 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, Exposed=Window]
    interface AuthenticatorAttestationResponse : AuthenticatorResponse {
        [SameObject] readonly attribute ArrayBuffer| attestationObject;
    }

    clientDataJSON
        This attribute, inherited from AuthenticatorResponse, contains
        the JSON serialized client data (see 6.3 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 the attestation
        statement 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 6.3
        Attestation, 5.3.4 Generating an Attestation Object, and Figure
        3.
    }

    5.2.2. Web Authentication Assertion (interface
AuthenticatorAssertionResponse)

    The AuthenticatorAssertionResponse interface represents an
    authenticator’s response to a client’s request for generation of a new
    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, Exposed=Window]
    interface AuthenticatorAssertionResponse : AuthenticatorResponse {
        [SameObject] readonly attribute ArrayBuffer| authenticatorData;
        [SameObject] readonly attribute ArrayBuffer| userHandle;
    }

    clientDataJSON
        This attribute, inherited from AuthenticatorResponse, contains
        the JSON serialized client data (see 5.1 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 6.1 Authenticator data.
    signature, of type ArrayBuffer, readonly
        This attribute contains the raw signature returned from the
        authenticator. See 6.2.3 The AuthenticatorGetAssertion
5.3. Parameters for Credential Generation (dictionary PublicKeyCredentialParameters)

```typescript
dictionary PublicKeyCredentialParameters {
  required PublicKeyCredentialType type;
  optional COSEAlgorithmIdentifier alg;
  required sequence<PublicKeyCredentialProperties> properties;
  optional listOf<PublicKeyCredentialExtension> extensions;
};
```

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 alg 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.

Note: we use "alg" as the latter member name, rather than spelling-out "algorithm", because it will be serialized into a message to the authenticator, which may be sent over a low-bandwidth link.

5.4. Options for Credential Creation (dictionary MakePublicKeyCredentialOptions)

```typescript
dictionary MakePublicKeyCredentialOptions {
  required PublicKeyCredentialRpEntity rp;
  required PublicKeyCredentialUserEntity user;
  required BufferSource challenge;
  required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
  optional unsigned long timeout;
  optional sequence<PublicKeyCredentialDescriptor> excludeCredentials = [];
  optional AuthenticatorSelectionCriteria authenticatorSelection;
  optional AuthenicationExtension<AuthenticatorExtensions> extensions;
};
```

rp, of type PublicKeyCredentialRpEntity

This member contains data about the Relying Party responsible for the request.

Its value's name member contains the friendly name of the Relying Party (e.g. "Acme Corporation", "Widgets, Inc.", or "Awesome Site").

Its value's id member specifies the relying party identifier with which the credential should be associated. If omitted, its value will be the CredentialsContainer object's relevant settings object's origin's effective domain.

user, of type PublicKeyCredentialUserEntity

This member contains data about the user account for which the Relying Party is requesting attestation.

Its value's name member contains a name for the user account (e.g., "John P. Smith").

Its value's id member contains the user handle for the account, Operation.

This attribute contains the user handle returned from the authenticator. See 6.2.2 The authenticatorGetAssertion operation.

5.3. Parameters for Credential Generation (dictionary PublicKeyCredentialParameters)

dictionary PublicKeyCredentialParameters {
  required PublicKeyCredentialType type;
  required COSEAlgorithmIdentifier alg;
  optional sequence<PublicKeyCredentialProperties> properties;
  optional listOf<PublicKeyCredentialExtension> extensions;
};
```

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 alg 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.

Note: we use "alg" as the latter member name, rather than spelling-out "algorithm", because it will be serialized into a message to the authenticator, which may be sent over a low-bandwidth link.

5.4. Options for Credential Creation (dictionary MakePublicKeyCredentialOptions)

dictionary MakePublicKeyCredentialOptions {
  required PublicKeyCredentialRpEntity rp;
  required PublicKeyCredentialUserEntity user;
  required BufferSource challenge;
  required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
  optional unsigned long timeout;
  optional sequence<PublicKeyCredentialDescriptor> excludeCredentials = [];
  optional AuthenticatorSelectionCriteria authenticatorSelection;
  optional AuthenticationExtensions<AuthenticatorExtensions> extensions;
};
```

rp, of type PublicKeyCredentialRpEntity

This member contains data about the Relying Party responsible for the request.

Its value's name member is required.

Its value's id member specifies the relying party identifier with which the credential should be associated. If omitted, its value will be the CredentialsContainer object's relevant settings object's origin's effective domain.

user, of type PublicKeyCredentialUserEntity

This member contains data about the user account for which the Relying Party is requesting attestation.

Its value's name, displayName and id members are required.
<table>
<thead>
<tr>
<th>Line</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>1806</td>
<td>challenge, of type BufferSource</td>
</tr>
<tr>
<td>1807</td>
<td>This member contains a challenge intended to be used for generating the newly created credential's attestation object.</td>
</tr>
<tr>
<td>1808</td>
<td>publicKeyCredParams, of type sequence&lt;PublicKeyCredentialParameters&gt;</td>
</tr>
<tr>
<td>1809</td>
<td>This member contains information about the desired properties of the credential to be created. The sequence is Thiered from most preferred to least preferred. The platform makes a best-effort to create the most preferred credential that it can.</td>
</tr>
<tr>
<td>1810</td>
<td>timeout, of type unsigned long</td>
</tr>
<tr>
<td>1811</td>
<td>This member 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.</td>
</tr>
<tr>
<td>1812</td>
<td>excludeCredentials, of type sequence&lt;PublicKeyCredentialDescriptor&gt;, defaulting to None</td>
</tr>
<tr>
<td>1813</td>
<td>This member is intended for use by Relying Parties that wish to limit the creation of multiple credentials for the same account on a single authenticator. The platform is requested to return an error if the new credential would be created on an authenticator that also contains one of the credentials enumerated in this parameter.</td>
</tr>
<tr>
<td>1814</td>
<td>authenticatorSelection, of type AuthenticatorSelectionCriteria</td>
</tr>
<tr>
<td>1815</td>
<td>This member is intended for use by Relying Parties that wish to select the appropriate authenticators to participate in the create() operation.</td>
</tr>
<tr>
<td>1816</td>
<td>attestation, of type AttestationConveyancePreference, defaulting to &quot;none&quot;</td>
</tr>
<tr>
<td>1817</td>
<td>This member is intended for use by Relying Parties that wish to express their preference for attestation conveyance. The default is none.</td>
</tr>
<tr>
<td>1818</td>
<td>extensions, of type AuthenticationExtensions</td>
</tr>
<tr>
<td>1819</td>
<td>This member 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. Some extensions are defined in 9 WebAuthn Extensions; consult the IANA &quot;WebAuthn Extension Identifier&quot; registry established by [WebAuthn-Registrations] for an up-to-date list of registered WebAuthn Extensions.</td>
</tr>
</tbody>
</table>

5.4.1. Public Key Entity Description (dictionary PublicKeyCredentialEntity)

The PublicKeyCredentialEntity dictionary describes a user account, or a Relying Party, with which a public key credential is associated.

```javascript
The PublicKeyCredentialEntity dictionary describes a user account, or a Relying Party, with which a public key credential is associated.
dictionary PublicKeyCredentialEntity {
  required DOMString name;
  USVString icon;
};
```

name, of type DOMString

A human-readable identifier for the entity. For example, this could be a company name for a Relying Party, or a user's name. This identifier is intended for display. Authenticators MUST accept and store a 64 byte minimum length for a name member's value. Authenticators MAY truncate a name member's value to a length equal to or greater than 64 bytes.

+ When inherited by PublicKeyCredentialRpEntity It is a human-friendly identifier for the Relying Party, intended only for display. For example, "ACME Corporation", "Wonderful Widgets, Inc." or "Awesome Site".
+ When inherited by PublicKeyCredentialUserEntity, It is a human-readable identifier for a user account. It is intended only for display, and SHOULD allow the user to easily tell the difference between user accounts with similar displayNames. For example, "alex", "alex.p.mueller@example.com" or
Authenticator Attachment enumeration (enum)

UserVerificationRequirement userVerification = "preferred";

dictionary AuthenticatorSelectionCriteria {
  AuthenticatorSelectionCriteria);

5.4.4. Authenticator Selection Criteria (dictionary)

Authenticators MUST accept and store a 128-byte minimum length
for an icon member's value. Authenticators MAY ignore an icon
member's value if its length is greater than 128 bytes.

5.4.2. RP Parameters for Credential Generation (dictionary)

The PublicKeyCredentialRpEntity dictionary is used to supply additional
Relying Party attributes when creating a new credential.

dictionary PublicKeyCredentialRpEntity : PublicKeyCredentialEntity {
  DOMString id;
  ...};

id, of type DOMString

A unique identifier for the Relying Party entity, which sets the
RP ID.

A serialized URL which resolves to an image associated with the
Relying Party's logo. This URL MUST be an a priori authenticated URL.

A serialized URL which resolves to an image associated with the
user. For example, this could be a user's avatar or a Relying
Party's logo. This URL MUST be an a priori authenticated URL.

5.4.3. User Account Parameters for Credential Generation (dictionary)

The PublicKeyCredentialUserEntity dictionary is used to supply additional
user account attributes when creating a new credential.

dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
  required BufferSource id;
  required DOMString displayName;
  ...};

id, of type BufferSource

The user handle of the user account entity.

displayName, of type DOMString

A human-friendly name for the user account, intended only for
display. For example, "Alex P. Miller" or "". The Relying
Party SHOULD let the user choose this, and SHOULD NOT restrict
the choice more than necessary.

Authenticators MUST accept and store a 64-byte minimum length
displayName member's value. Authenticators MAY truncate a
displayName member's value to a length equal to or greater than 64 bytes.

5.4.4. Authenticator Selection Criteria (dictionary)

AuthenticatorSelectionCriteria

Relying Parties may use the AuthenticatorSelectionCriteria dictionary
to specify their requirements regarding authenticator attributes.

dictionary AuthenticatorSelectionCriteria {
  AuthenticatorAttachment authenticatorAttachment;
  boolean requireResidentKey = false;
  UserVerificationRequirement userVerification = "preferred";
};

authenticatorAttachment, of type AuthenticatorAttachment

If this member is present, eligible authenticators are filtered
to only authenticators attached with the specified 5.4.5
Authenticator Attachment enumeration (enum

Authenticator Attachment enumeration (enum

Authenticators MUST accept and store a 64-byte minimum length
displayName member's value. Authenticators MAY truncate a
displayName member's value to a length equal to or greater than 64 bytes.

Authenticators MUST accept and store a 64-byte minimum length
for a name member's value. Authenticators MAY truncate a name
member's value to a length equal to or greater than 64 bytes.

5.4.2. RP Parameters for Credential Generation (dictionary)

The PublicKeyCredentialRpEntity dictionary is used to supply additional
Relying Party attributes when creating a new credential.

dictionary PublicKeyCredentialRpEntity : PublicKeyCredentialEntity {
  DOMString id;
  ...};

id, of type DOMString

A unique identifier for the Relying Party entity, which sets the
RP ID.

A human-friendly username account identifier to the name
member of PublicKeyCredentialUserEntity.

Authenticators MUST accept and store a 64-byte minimum length
displayName member's value. Authenticators MAY truncate a
displayName member's value to a length equal to or greater than 64 bytes.

5.4.4. Authenticator Selection Criteria (dictionary

AuthenticatorSelectionCriteria)

Relying Parties may use the AuthenticatorSelectionCriteria dictionary
to specify their requirements regarding authenticator attributes.

dictionary AuthenticatorSelectionCriteria {
  AuthenticatorAttachment authenticatorAttachment;
  boolean requireResidentKey = false;
  UserVerificationRequirement userVerification = "preferred";
};

authenticatorAttachment, of type AuthenticatorAttachment

If this member is present, eligible authenticators are filtered
to only authenticators attached with the specified 5.4.5
Authenticator Attachment enumeration (enum

Authenticator Attachment enumeration (enum

Authenticators MUST accept and store a 64-byte minimum length
displayName member's value. Authenticators MAY truncate a
displayName member's value to a length equal to or greater than 64 bytes.

Authenticators MUST accept and store a 64-byte minimum length
for a name member's value. Authenticators MAY truncate a name
member's value to a length equal to or greater than 64 bytes.

Icon, of type USVString

A serialized URL which resolves to an image associated with the
entity. For example, this could be a Relying Party's logo.

A serialized URL which resolves to an image associated with the
entity. For example, this could be a Relying Party's logo.

Authenticators MUST accept and store a 64-byte minimum length
for an icon member's value. Authenticators MAY ignore an icon
member's value if its length is greater than 128 bytes.

Display Name member's value to a length equal to or greater than
64 bytes.

A unique identifier for the Relying Party entity, which sets the
Relying Party's logo. This URL MUST be an a priori authenticated URL.

A serialized URL which resolves to an image associated with the
user. For example, this could be a user's avatar or a Relying
Party's logo. This URL MUST be an a priori authenticated URL.

Authenticators MUST accept and store a 64-byte minimum length
for an icon member's value. Authenticators MAY ignore an icon
member's value if its length is greater than 128 bytes.
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 cross-platform transport protocols such as Bluetooth (see 5.8.4) to discover and communicate with cross-platform attached authenticators. Therefore, we use AuthenticatorAttachment to describe an authenticator’s attachment modality. We define authenticators that are attached using cross-platform transports. Usually, authenticators of this 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 may be created using platform-specific transports. Usually, authenticators of this class are removable from, and can “roam” among, client platforms. * 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 may be created using platform-specific transports. Usually, authenticators of this class are removable from, and can “roam” among, client platforms. * 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.

Clients can 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 cross-platform transport protocols such as Bluetooth (see 5.10.4) to discover and communicate with cross-platform attached authenticators. Therefore, we use AuthenticatorAttachment 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 platform-specific transports. Usually, authenticators of this class are non-removable from the platform. A public key credential bound to a platform authenticator is called a platform credential. * 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. A public key credential bound to a roaming authenticator is called a roaming credential.

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 may be created using platform-specific transports. Usually, authenticators of this class are removable from, and can “roam” among, client platforms. * 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.

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 cross-platform transport protocols such as Bluetooth (see 5.8.4) to discover and communicate with cross-platform attached authenticators. Therefore, we use AuthenticatorAttachment 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 platform-specific transports. Usually, authenticators of this class are non-removable from the platform. A public key credential bound to a platform authenticator is called a platform credential. * 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. A public key credential bound to a roaming authenticator is called a roaming credential.

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 may be created using platform-specific transports. Usually, authenticators of this class are removable from, and can “roam” among, client platforms. * 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.

Note: An attachment modality selection option is available only in the CreateUser([origin, options], sameOriginWithAncestors) API. The Relying Party may use it to, for example, ensure the user has a roaming credential for authenticating using other clients; or to specifically discover a platform credential for easier reauthentication of a particular client. The ([DiscoverFromExternalSource][origin, options], sameOriginWithAncestors) operation has no attachment modality selection option, so the Relying Party should accept any of the user’s registered credentials. The client and user will then use whichever is available and convenient at the time.

5.4.6. Attestation Conveyance Preference enumeration (enum AuthenticatorAttachment).
This optional member contains a list of defaulting to None PublicKeyCredentialDescriptor objects representing public key credentials acceptable to the caller, in descending order of the origin's effective domain.

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.

dictionary PublicKeyCredentialRequestOptions {
  required BufferSource challenge;
  optional unsigned long timeout;
  optional USVString rpId;
  optional sequence<PublicKeyCredentialDescriptor> allowCredentials = [];
  optional UserVerificationRequirement userVerification = "preferred";
  optional AuthenticationExtensions extensions;
};

challenge, of type BufferSource
This member represents a challenge that the selected authenticator signs, along with other data, when producing an authentication assertion. See the 13.1 Cryptographic Challenges security consideration.

timeout, of type unsigned long
This optional member specifies a time, in milliseconds, that the caller is willing to wait for the call to complete. The value is treated as a hint, and may be overridden by the platform.

rpId, of type USVString
This optional member specifies the relying party identifier claimed by the caller. If omitted, its value will be the CredentialsContainer object's relevant settings object's origin's effective domain.

allowCredentials, of type sequence<PublicKeyCredentialDescriptor>, defaulting to None
This optional member contains a list of PublicKeyCredentialDescriptor objects representing public key credentials acceptable to the caller, in descending order of

"none", "indirect", "direct";

* none - indicates that the Relying Party is not interested in authenticator attestation. The client may replace the AAGUID and attestation statement generated by the authenticator with meaningless client-generated values. For example, in order to avoid having to obtain user consent to relay unique user information to the Relying Party, or to save a roundtrip to a Privacy CA.

This is the default value.

* indirect - indicates that the Relying Party prefers an attestation conveyance yielding verifiable attestation statements, but allows the client to decide how to obtain such attestation statements. The client MAY replace the authenticator-generated attestation statements with attestation statements generated by a Privacy CA, in order to protect the user's privacy, or to assist Parties with attestation verification in a heterogeneous ecosystem.

Note: There is no guarantee that the Relying Party will obtain a verifiable attestation statement in this case. For example, in the case that the authenticator employs self attestation,

* direct - indicates that the Relying Party wants to receive the attestation statement as generated by the authenticator.

5.5. Options for Assertion Generation (dictionary

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.

dictionary PublicKeyCredentialRequestOptions {
  required BufferSource challenge;
  optional unsigned long timeout;
  optional USVString rpId;
  optional sequence<PublicKeyCredentialDescriptor> allowCredentials = [];
  optional UserVerificationRequirement userVerification = "preferred";
  optional AuthenticationExtensions extensions;
};

challenge, of type BufferSource
This member represents a challenge that the selected authenticator signs, along with other data, when producing an authentication assertion. See the 13.1 Cryptographic Challenges security consideration.

timeout, of type unsigned long
This optional member specifies a time, in milliseconds, that the caller is willing to wait for the call to complete. The value is treated as a hint, and MAY be overridden by the platform.

rpId, of type USVString
This optional member specifies the relying party identifier claimed by the caller. If omitted, its value will be the CredentialsContainer object's relevant settings object's origin's effective domain.

allowCredentials, of type sequence<PublicKeyCredentialDescriptor>, defaulting to None
This optional member contains a list of PublicKeyCredentialDescriptor objects representing public key credentials acceptable to the caller, in descending order of

"none", "indirect", "direct";

* none - indicates that the Relying Party is not interested in authenticating the client. The client may replace the AAGUID and attestation statement generated by the authenticator with meaningless client-generated values. For example, in order to avoid having to obtain user consent to relay identifying information to the Relying Party, or to save a roundtrip to an Attestation CA.

This is the default value.

* indirect - indicates that the Relying Party prefers an attestation conveyance yielding verifiable attestation statements, but allows the client to decide how to obtain such attestation statements. The client MAY replace the authenticator-generated attestation statements with attestation statements generated by an Anonymization CA, in order to protect the user's privacy, or to assist Parties with attestation verification in a heterogeneous ecosystem.

Note: There is no guarantee that the Relying Party will obtain a verifiable attestation statement in this case. For example, in the case that the authenticator employs self attestation,

* direct - indicates that the Relying Party wants to receive the attestation statement as generated by the authenticator.
5.8. Authentication Extensions

Developers are encouraged to leverage the `AbortController` to manage the abort of the `[[Create]](origin, options, sameOriginWithAncestors)` and `[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)` operations. See DOM 3.3 Using AbortController and AbortSignal objects in APIs section for detailed instructions.

Note: DOM 3.3 Using AbortController and AbortSignal objects in APIs section specifies that web platform APIs integrating with the `Credential Management 1 2.5.4 Create a Credential immediately before calling [[Create]](origin, options, sameOriginWithAncestors), then in 5.1.3 Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method right before authenticator sessions start, and finally during authenticator sessions start, and finally during authenticator sessions start. The same goes for `[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)`, `[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)`, and `[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)` operations should be aborted.

The visibility and focus state of the Window object determines whether the `[[Create]](origin, options, sameOriginWithAncestors)` and `[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)` operations should continue. When the Window object associated with the [Document loses focus, [[Create]](origin, options, sameOriginWithAncestors) operations should continue. When the Window object associated with the [Document loses focus, [[Create]](origin, options, sameOriginWithAncestors) and `[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)` operations should continue. When the Window object associated with the [Document loses focus, [[Create]](origin, options, sameOriginWithAncestors) and `[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)` operations should be aborted.

The WHATWG HTML WG is discussing whether to provide a hook when a browsing context gains or loses focus. If a hook is provided, this member will be updated to include the hook. See WHATWG HTML WG issue #2711 for more details.

5.8. Extension Data Structures

### 5.8.1. Extension Data Structures

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.

5.8.2. Extension Data Structures

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.

5.8.3. Extension Data Structures

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.

5.8.4. Extension Data Structures

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.

5.8.5. Extension Data Structures

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.

5.8.6. Extension Data Structures

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.

5.8.7. Extension Data Structures

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.

5.8.8. Extension Data Structures

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

5.8.1. 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
[interface CollectedClientData {
    type: required DOMString;
    challenge: required DOMString;
    origin: required DOMString;
    hashAlgorithm: DOMString;
    tokenBindingId: DOMString;
    authenticatorExtensions: AuthenticationExtensions;
    clientExtensions: AuthenticationExtensions;
};]
```

The type member contains the string "webauthn.create" when creating new credentials, and "webauthn.get" when getting an assertion from an existing credential. The purpose of this member is to prevent certain types of signature confusion attacks (where an attacker substitutes one legitimate signature for another).

The challenge member contains the base64url encoding of the challenge provided by the RP. See the 13.1 Cryptographic Challenges security consideration.

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

The hashAlgorithm member is a recognized algorithm name that supports the "digest" operation, which specifies the algorithm used to compute the hash of the serialized client data. This algorithm is chosen by the client at its sole discretion.

The tokenBindingId 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 clientExtensions and authenticatorExtensions members contain additional parameters generated by processing the extensions passed in by the Relaying Party. WebAuthn extensions are detailed in Section 9 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.
- Hash of the serialized client data
- This is the hash (computed using SHA-256) of the JSON-serialized client data, as constructed by the client.

### 5.10.4. Authenticator Transport enumeration (enum AuthenticatorTransport)

- `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 (Bluetooth Low Energy / BLE).

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".

### 5.10.3. Credential Descriptor (dictionary PublicKeyCredentialDescriptor)

- required PublicKeyCredentialType `type`;
- required BufferSource `id`;
- sequence<AuthenticatorTransport> `transports`;

This dictionary contains the attributes that are specified by a caller when referring to a credential as an input parameter to the `create()` or `get()` methods. It mirrors the fields of the PublicKeyCredential object returned by the latter 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.

### 5.10.5. Cryptographic Algorithm Identifier (typedef COSEAlgorithmIdentifier)

Supported cryptographic algorithms are standard for COSE.

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 (Bluetooth Low Energy / BLE).

### 5.8.5. Cryptographic Algorithm Identifier (typedef COSEAlgorithmIdentifier)

typedef long COSEAlgorithmIdentifier;
A COSEAlgorithmIdentifier's value is a number identifying a cryptographic algorithm. The algorithm identifiers SHOULD be values registered in the IANA COSE Algorithms registry [IANA-COSE-ALGS-REG], for instance, -7 for "ES256" and -257 for "RS256".

5.8.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

enum UserVerificationRequirement {
    "required",
    "preferred",
    "discouraged"
};

A Relying Party may require user verification for some of its operations but not for others, and may use this type to express its needs.

The value required indicates that the Relying Party requires user verification for the operation and will fail the operation if the response does not have the UV flag set.

The value preferred indicates that the Relying Party prefers user verification for the operation if possible, but will fail the operation if the response does not have the UV flag set.

The value discouraged indicates that the Relying Party does not want user verification employed during the operation (e.g., in the interest of minimizing disruption to the user interaction flow).

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 the Web Authentication API.

6. WebAuthn Authenticator model

The Web Authentication API implies a specific abstract functional model for an authenticator. This section describes that authenticator model.

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, this abstract model 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 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 housed in a separate device entirely. The authenticator itself can contain a cryptographic module which operates at a higher security level than the rest of the authenticator. This is particularly

6. WebAuthn Authenticator model

The Web Authentication API implies a specific abstract functional model for an authenticator. This section describes that authenticator model.

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, this abstract model 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 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 housed in a separate device entirely. The authenticator itself can contain a cryptographic module which operates at a higher security level than the rest of the authenticator. This is particularly

6. WebAuthn Authenticator model

The Web Authentication API implies a specific abstract functional model for an authenticator. This section describes that authenticator model.

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, this abstract model 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 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 housed in a separate device entirely. The authenticator itself can contain a cryptographic module which operates at a higher security level than the rest of the authenticator. This is particularly

6. WebAuthn Authenticator model

The Web Authentication API implies a specific abstract functional model for an authenticator. This section describes that authenticator model.

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, this abstract model 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 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 housed in a separate device entirely. The authenticator itself can contain a cryptographic module which operates at a higher security level than the rest of the authenticator. This is particularly

6. WebAuthn Authenticator model

The Web Authentication API implies a specific abstract functional model for an authenticator. This section describes that authenticator model.

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, this abstract model 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 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 housed in a separate device entirely. The authenticator itself can contain a cryptographic module which operates at a higher security level than the rest of the authenticator. This is particularly

6. WebAuthn Authenticator model

The Web Authentication API implies a specific abstract functional model for an authenticator. This section describes that authenticator model.

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, this abstract model 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 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 housed in a separate device entirely. The authenticator itself can contain a cryptographic module which operates at a higher security level than the rest of the authenticator. This is particularly

6. WebAuthn Authenticator model

The Web Authentication API implies a specific abstract functional model for an authenticator. This section describes that authenticator model.

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, this abstract model 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 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 housed in a separate device entirely. The authenticator itself can contain a cryptographic module which operates at a higher security level than the rest of the authenticator. This is particularly

6. WebAuthn Authenticator model

The Web Authentication API implies a specific abstract functional model for an authenticator. This section describes that authenticator model.

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, this abstract model 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 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 housed in a separate device entirely. The authenticator itself can contain a cryptographic module which operates at a higher security level than the rest of the authenticator. This is particularly

6. WebAuthn Authenticator model

The Web Authentication API implies a specific abstract functional model for an authenticator. This section describes that authenticator model.

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, this abstract model 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 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 housed in a separate device entirely. The authenticator itself can 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, for example, be a TPM) could be considered more trustworthy than the rest of the authenticator.

Each authenticator can have some number of public key credentials. Each public key credential has an identifier which is unique (or extremely unlikely to be duplicated) among all public key credentials. Each public credential is also associated with a Relaying Party, whose identity is represented by a Relaying Party Identifier (RP ID).

Each authenticator has an AAGUID, which is a 128-bit identifier that identifies 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 all other types of authenticators. The RP MAY use the AAGUID to infer certain properties of the 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 signatures are observed and added at different levels of the stack as a signature, per the attestation procedure, is sent from the client platform to the RP. The authenticator checks these bindings against expected values. These contextual bindings are divided into two parts: 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 data and sends only those that are necessary to the result to the authenticator. The authenticator signs over the combination of the hash of the serialized client data, 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 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 public key credential is created via an authenticatorMakeCredential operation. An attestation signature provides cryptographic proof of certain properties of the the authenticator and the credential. For instance, an attestation signature asserts the type of the authenticator (as denoted by its AAGUID) and the credential public key. The attestation signature is signed by an authenticator's attestation private key, which is chosen depending on the type of attestation desired. For more details on attestation, see 6.3.1 Attestation.

2. An assertion signature is produced when 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. The assertion signature asserts that the authenticator possesses a private key that has been created by the user, and that the user possesses a particular credential private key has established, to the best of its ability, that the user requesting this transaction is the same user who consented to creating that particular public key credential. It also asserts additional information, termed client data, that may be useful to the caller, such as the means by which the user requested this transaction.

Additionally, each authenticator has an AAGUID, which is a 128-bit identifier indicating the type 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 all other types of authenticators. The RP MAY use the AAGUID to infer certain properties of the 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 signatures are observed and added at different levels of the stack as a signature, per the attestation procedure, is sent from the client platform to the RP. The authenticator checks these bindings against expected values. These contextual bindings are divided into two parts: 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 data and sends only those that are necessary to the result to the authenticator. The authenticator signs over the combination of the hash of the serialized client data, 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 public key credential is created via an authenticatorMakeCredential operation. An attestation signature provides cryptographic proof of certain properties of the the authenticator and the credential. For instance, an attestation signature asserts the type of the authenticator (as denoted by its AAGUID) and the credential public key. The attestation signature is signed by an authenticator's attestation private key, which is chosen depending on the type of attestation desired. For more details on attestation, see 6.3.1 Attestation.

2. An assertion signature is produced when 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. The assertion signature asserts that the authenticator possesses a private key that has been created by the user, and that the user possesses a particular credential private key has established, to the best of its ability, that the user requesting this transaction is the same user who consented to creating that particular public key credential. It also asserts additional information, termed client data, that may be useful to the caller, such as the means by which the user requested this transaction.
which user consent was provided, and the prompt shown to the user by the authenticator. The assertion signature format is illustrated in Figure 2, below.

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

6.1. Authenticator data

The authenticator data structure encodes contextual bindings made by the authenticator. These bindings are controlled by the authenticator itself, and derive their trust from the Relying Party’s assessment of the authenticity 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 data. At 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 authenticator data structure is a byte array of 37 bytes or more, as follows.

<table>
<thead>
<tr>
<th>Name Length (in bytes) Description</th>
<th>rpIdHash 32 SHA-256 hash of the RP ID associated with the credential.</th>
<th>flags 1 Flags (bit 0 is the least significant bit):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0: User Present (UP) result.</td>
<td>+ 1 means the user is present.</td>
<td>Bit 1: Reserved for future use (RFU1).</td>
</tr>
<tr>
<td>+ 0 means the user is not present.</td>
<td>+ 1 means the user is verified.</td>
<td>Bit 2: User Verified (UV) result.</td>
</tr>
<tr>
<td>* Bit 1: Reserved for future use (RFU1).</td>
<td>+ 0 means the user is not verified.</td>
<td>+ 1 means the user is verified.</td>
</tr>
<tr>
<td>* Bit 2: User Verified (UV) result.</td>
<td>+ 0 means the user is not verified.</td>
<td>+ 0 means the user is not verified.</td>
</tr>
<tr>
<td>* Bit 3: 3-5: Reserved for future use (RFU2).</td>
<td>+ 1 means the user is verified.</td>
<td>+ 1 means the user is verified.</td>
</tr>
<tr>
<td>* Bit 6: Attested credential data included (AT).</td>
<td>+ Indicates whether the authenticator added attested credential data.</td>
<td>* Bit 7: Extension data included (ED).</td>
</tr>
<tr>
<td>+ Indicates if the authenticator data has extensions.</td>
<td>+ Indicates if the authenticator data has extensions.</td>
<td>ExtensionData variable (if present) Extension-defined authenticator data.</td>
</tr>
<tr>
<td>signCount 4 Signature counter, 32-bit unsigned big-endian integer.</td>
<td>attestedCredentialData variable (if present) attested credential data (if present). See 6.3.1 Attested credential data for details. Its length depends on the length of the credential ID and credential public key being attested.</td>
<td>This is a CBOR [RFC7049] map with extension identifiers as keys, and authenticator extension outputs as values. See 9 WebAuthn Extensions for details.</td>
</tr>
</tbody>
</table>

NOTE: The names in the Name column in the above table are only for reference within this document, and are not present in the actual representation of the authenticator data.

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, by verifying that the RP ID associated with the requested credential exactly matches the RP ID supplied by the client, and that the RP ID is a registrable domain suffix or is equal to the effective domain of the RP’s origin’s effective domain.
The UP flag SHALL be set if and only if the authenticator detected a user through an authenticator specific gesture. The RU flag SHALL be set to zero.

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

If the authenticator does not include any extension data, it MUST set the ED flag to zero, and to one if extension data is included.

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

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 attested credential 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 attested credential data, plus the length of the CBOR map that follows.

6.1.1. Signature Counter Considerations

Authenticators MUST implement a signature counter feature. The signature counter is incremented for each successful authenticatorGetAssertion operation by some positive value, and its value is returned to the Relying Party within the authenticator data.

The signature counter’s purpose is to aid Relying Parties in detecting cloned authenticators. Clone detection is more important for authenticators with limited protection measures.

An Relying Party stores the signature counter of the most recent authenticatorGetAssertion operation. Upon a new authenticatorGetAssertion operation, the Relying Party compares the stored signature counter value with the new signCount value returned in the assertion’s authenticator data. If this new signCount value is less than or equal to the stored value, a cloned authenticator may exist, or the authenticator may be malfunctioning.

Detecting a signature counter mismatch does not indicate whether the current operation was performed by a cloned authenticator or the original authenticator. Relying Parties should address this situation appropriately relative to their individual situations, i.e., their risk tolerance.

Authenticators:
* should implement per-RP ID signature counters. This prevents the signature counter value from being shared between Relying Parties and being possibly employed as a correlation handle for the user.
* should ensure that the signature counter value does not accidentally decrease (e.g., due to hardware failures).

6.2. Authenticator operations

A WebAuthn 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 maintains 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.

6.2.1. The authenticatorMakeCredential operation

The UP flag SHALL be set if and only if the authenticator detected a user through an authenticator specific gesture. The RU flag SHALL be set to zero.

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

If the authenticator does not include any extension data, it MUST set the ED flag to zero, and to one if extension data is included.

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

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 attested credential 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 attested credential data, plus the length of the CBOR map that follows.

The signature counter's purpose is to aid Relying Parties in detecting cloned authenticators. Clone detection is more important for authenticators with limited protection measures.

An Relying Party stores the signature counter of the most recent authenticatorGetAssertion operation. Upon a new authenticatorGetAssertion operation, the Relying Party compares the stored signature counter value with the new signCount value returned in the assertion's authenticator data. If this new signCount value is less than or equal to the stored value, a cloned authenticator may exist, or the authenticator may be malfunctioning.

Detecting a signature counter mismatch does not indicate whether the current operation was performed by a cloned authenticator or the original authenticator. Relying Parties should address this situation appropriately relative to their individual situations, i.e., their risk tolerance.

Authenticators:
* should implement per-RP ID signature counters. This prevents the signature counter value from being shared between Relying Parties and being possibly employed as a correlation handle for the user.
* should ensure that the signature counter value does not accidentally decrease (e.g., due to hardware failures).

6.2. Authenticator operations

A WebAuthn 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 maintains 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.

6.2.1. The authenticatorMakeCredential operation

The UP flag SHALL be set if and only if the authenticator detected a user through an authenticator specific gesture. The RU flag SHALL be set to zero.

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

If the authenticator does not include any extension data, it MUST set the ED flag to zero, and to one if extension data is included.

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

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 attested credential 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 attested credential data, plus the length of the CBOR map that follows.

The signature counter's purpose is to aid Relying Parties in detecting cloned authenticators. Clone detection is more important for authenticators with limited protection measures.

An Relying Party stores the signature counter of the most recent authenticatorGetAssertion operation. Upon a new authenticatorGetAssertion operation, the Relying Party compares the stored signature counter value with the new signCount value returned in the assertion's authenticator data. If this new signCount value is less than or equal to the stored value, a cloned authenticator may exist, or the authenticator may be malfunctioning.

Detecting a signature counter mismatch does not indicate whether the current operation was performed by a cloned authenticator or the original authenticator. Relying Parties should address this situation appropriately relative to their individual situations, i.e., their risk tolerance.

Authenticators:
* should implement per-RP ID signature counters. This prevents the signature counter value from being shared between Relying Parties and being possibly employed as a correlation handle for the user.
* should ensure that the signature counter value does not accidentally decrease (e.g., due to hardware failures).
It takes the following input parameters:

- **hash**: The hash of the serialized client data, provided by the client.
- **rpEntity**: The Relying Party’s PublicKeyCredentialRpEntity.
- **userEntity**: The user account’s PublicKeyCredentialUserEntity, containing the user handle given by the Relying Party.
- **requireResidentKey**: The authenticatorSelection.requireResidentKey value given by the Relying Party.
- **requireUserPresence**: A Boolean value provided by the client, which in invocations from a WebAuthn Client’s ([Create]([origin, options, sameOriginWithAncestors]) method is always set to the inverse of requireUserVerification.
- **requireUserVerification**: The effective user verification requirement for credential creation, a Boolean value provided by the client.
- **credTypesAndPubKeyAlgs**: A sequence of pairs of PublicKeyCredentialType and public key algorithms (COSEAlgorithmIdentifier) requested by the Relying Party. This sequence is ordered from most preferred to least preferred. The platform makes a best-effort to create the most preferred credential that it can.
- **excludeCredentialDescriptorList**: An optional list of PublicKeyCredentialDescriptor 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. excludeCredentialDescriptorList contains a list of known credentials.
- **extensions**: A map from extension identifiers to their authenticator extension inputs, created by the client based on the extensions requested by the Relying Party, if any. Note: Before performing this operation, all other operations in progress in the authenticator session must be aborted by running the authenticatorCancel operation.

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

1. 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.
2. Check if at least one of the specified combinations of PublicKeyCredentialType and cryptographic parameters in credTypesAndPubKeyAlgs is valid.

The result of looking up a credential id credentialId in an authenticator authenticator is the result of the following algorithm:

1. If authenticator can decrypt credentialId into a public key credential source credSource:
   1. Set credSource.id to credentialId.
   2. Return credSource.
2. For each public key credential source credSource of authenticator’s credentials:
   1. If credSource.id is credentialId, return credSource.
3. Return null.

### 6.2.2. The authenticatorMakeCredential operation

It takes the following input parameters:

- **hash**: The hash of the serialized client data, provided by the client.
- **rpEntity**: The Relying Party’s PublicKeyCredentialRpEntity.
- **userEntity**: The user account’s PublicKeyCredentialUserEntity, containing the user handle given by the Relying Party.
- **requireResidentKey**: The authenticatorSelection.requireResidentKey value given by the Relying Party.
- **requireUserPresence**: A Boolean value provided by the client, which in invocations from a WebAuthn Client’s ([Create]([origin, options, sameOriginWithAncestors]) method is always set to the inverse of requireUserVerification.
- **requireUserVerification**: The effective user verification requirement for credential creation, a Boolean value provided by the client.
- **credTypesAndPubKeyAlgs**: A sequence of pairs of PublicKeyCredentialType and public key algorithms (COSEAlgorithmIdentifier) requested by the Relying Party, if any. These are known to the authenticator, it should not create a new credential. excludeCredentialDescriptorList contains a list of known credentials.
- **extensions**: A CBOR map from extension identifiers to their authenticator extension inputs, created by the client based on the extensions requested by the Relying Party, if any. Note: Before performing this operation, all other operations in progress in the authenticator session MUST be aborted by running the authenticatorCancel operation.

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

1. 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.
2. Check if at least one of the specified combinations of PublicKeyCredentialType and cryptographic parameters in credTypesAndPubKeyAlgs is valid.
4. If requireUserVerification is true and the authenticator cannot perform user verification, return an error code equivalent to "ConstraintError" and terminate the operation.

5. If requireUserVerification is true and the authenticator cannot perform user verification, return an error code equivalent to "ConstraintError" and terminate the operation.

6. Obtain user consent for creating a new credential. The prompt for obtaining this consent is shown by the authenticator if it has its own output capability, or by the user agent otherwise. The prompt SHOULD display rpEntity.id, rpEntity.name, userEntity.name and userEntity.displayName, if possible.

7. If requireUserVerification is true, the method of obtaining user consent MUST include user verification.

8. If requirePresence is true, the method of obtaining user consent MUST include a test of user presence.

9. If the user denies consent or if user verification fails, return an error code equivalent to "NotAllowedError" and terminate the operation.

10. Once user consent has been obtained, generate a new credential object:

    1. Let (publicKey, privateKey) be a new pair of cryptographic keys using the combination of PublicKeyCredentialType and cryptographic parameters represented by the first item in credTypesAndPublicKeyAlgs that is supported by this authenticator.

    2. Let credentialId be a new identifier for this credential that is globally unique with high probability across all credentials with the same type across all authenticators.

    3. Let userHandle be userEntity.id.

    4. Associate the credentialId and private key with rpEntity.id and userHandle.

    5. If excludeCredentialDescriptorList.type matches rpEntity.id and an excludeCredentialDescriptorList item's credentialSource is the same as this authenticator, return an error code equivalent to "NotAllowedError" and terminate the operation.

    6. If requireResidentKey is true and the authenticator cannot store a resident key, return an error code equivalent to "NotAllowedError" and terminate the operation.

    7. If requireResidentKey is true and the authenticator cannot store a resident key, return an error code equivalent to "NotAllowedError" and terminate the operation.

    8. If the user denies consent or if user verification fails, return an error code equivalent to "ConstraintError" and terminate the operation.

    9. Obtain user consent for creating a new credential. The prompt for obtaining this consent is shown by the authenticator if it has its own output capability, or by the user agent otherwise. The prompt SHOULD display rpEntity.id, rpEntity.name, userEntity.name and userEntity.displayName, if possible.

10. If requireUserVerification is true, the method of obtaining user consent MUST include user verification.

11. If requirePresence is true, the method of obtaining user consent MUST include a test of user presence.

12. If the user does not consent or if user verification fails, return an error code equivalent to "NotAllowedError" and terminate the operation.

13. Once user consent has been obtained, generate a new credential object:

    1. Let (publicKey, privateKey) be a new pair of cryptographic keys using the combination of PublicKeyCredentialType and cryptographic parameters represented by the first item in credTypesAndPublicKeyAlgs that is supported by this authenticator.

    2. Let credentialId be a new identifier for this credential that is globally unique with high probability across all credentials with the same type across all authenticators.

    3. Let userHandle be userEntity.id.

    4. Associate the credentialId and private key with rpEntity.id and userHandle.

    5. Delete any older credentials with the same rpEntity.id and userHandle that are stored locally by the authenticator.

14. If requireResidentKey is true and the authenticator cannot store a resident key, return an error code equivalent to "NotAllowedError" and terminate the operation.

15. If requireResidentKey is true and the authenticator cannot store a resident key, return an error code equivalent to "NotAllowedError" and terminate the operation.

16. If the user denies consent or if user verification fails, return an error code equivalent to "ConstraintError" and terminate the operation.

17. Obtain user consent for creating a new credential. The prompt for obtaining this consent is shown by the authenticator if it has its own output capability, or by the user agent otherwise. The prompt SHOULD display rpEntity.id, rpEntity.name, userEntity.name and userEntity.displayName, if possible.

18. If requireUserVerification is true, the method of obtaining user consent MUST include user verification.

19. If requirePresence is true, the method of obtaining user consent MUST include a test of user presence.

20. If the user does not consent or if user verification fails, return an error code equivalent to "NotAllowedError" and terminate the operation.

21. Confirm that consent to create a new credential has been obtained:

    1. If looking up descriptorList.id in this authenticator returns null, and the returned item's RP ID and type match rpEntity.id and includeCredentialDescriptorList.type respectively, then obtain user consent for creating a new credential. The method of obtaining user consent MUST include a test of user presence. If the user does not consent or if user verification fails, return an error code equivalent to "NotAllowedError" and terminate the operation.

    2. If descriptor in descriptorList is null, and the returned item's RP ID and type match rpEntity.id and excludeCredentialDescriptorList.type respectively, then obtain user consent for creating a new credential. The method of obtaining user consent MUST include a test of user presence. If the user does not consent or if user verification fails, return an error code equivalent to "NotAllowedError" and terminate the operation.

    3. For each descriptor of includeCredentialDescriptorList:

        1. If looking up descriptor.id in this authenticator returns non-null, and the returned item's RP ID and type match rpEntity.id and includeCredentialDescriptorList.type respectively, then obtain user consent for creating a new credential. The method of obtaining user consent MUST include a test of user presence. If the user does not consent or if user verification fails, return an error code equivalent to "NotAllowedError" and terminate the operation.

    4. For each descriptor of excludeCredentialDescriptorList:

        1. If looking up descriptor.id in this authenticator returns non-null, and the returned item's RP ID and type match rpEntity.id and excludeCredentialDescriptorList.type respectively, then obtain user consent for creating a new credential. The method of obtaining user consent MUST include a test of user presence. If the user does not consent or if user verification fails, return an error code equivalent to "NotAllowedError" and terminate the operation.

22. Once user consent has been obtained, generate a new credential object:

    1. Let (publicKey, privateKey) be a new pair of cryptographic keys using the combination of PublicKeyCredentialType and cryptographic parameters represented by the first item in credTypesAndPublicKeyAlgs that is supported by this authenticator.

    2. Let credentialId be a new identifier for this credential that is globally unique with high probability across all credentials with the same type across all authenticators.

    3. Let userHandle be userEntity.id.

    4. Associate the credentialId and private key with rpEntity.id and userHandle.

    5. Delete any older credentials with the same rpEntity.id and userHandle that are stored locally by the authenticator.
8. If any error occurred while creating the new credential object, return an error code equivalent to "UnknownError" and terminate the operation.

9. Let processedExtensions be the result of authenticator extension processing for each supported extension identifier/input pair in extensions.

10. If the authenticator supports:

   a per-RP ID signature counter
   allocate the counter, associate it with the RP ID, and
   initialize the counter value as zero.

   a global signature counter
   Use the global signature counter's actual value when
   generating authenticator data.

   a per credential signature counter
   allocate the counter, associate it with the new
   credential, and initialize the counter value as zero.

11. Let attestedCredentialData be the attested credential data byte
   array including the credentialId and publicKey.

12. Let authenticatorData be the byte array specified in 6.1
   Authenticator data, including attestedCredentialData as the
   attestedCredentialData and processedExtensions, if any, as the
   extensions.

13. Return the attestation object for the new credential created by the
   procedure specified in 6.3.4 Generating an Attestation Object
   using an authenticator-chosen attestation statement format,
   authenticatorData, and hash. For more details on attestation, see
   6.3 Attestation.

On successful completion of this operation, the authenticator returns
the attestation object to the client.

6.2.2. The authenticatorGetAssertion operation

It takes the following input parameters:

- rpId
  The caller's RP ID, as determined by the user agent and the
  client.

- hash
  The hash of the serialized client data, provided by the client.

allowCredentialDescriptorList
An optional list of PublicKeyCredentialDescriptors describing
credentials acceptable to the Relying Party (possibly filtered
by the client), if any.

requireUserPresence
A Boolean value provided by the client, which in invocations
from a WebAuthn Client's [GetAssertionFromExternalSource][origin,
options, sameOriginWithAncestors] method is always set to the
inverse of requireUserVerification.

requireUserVerification
The effective user verification requirement for assertion, a
Boolean value provided by the client.

extensions
A map from extension identifiers to their authenticator
extension inputs, created by the client based on the extensions
requested by the Relying Party, if any.
When this method is invoked, the authenticator **must** perform the following procedure:

1. 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.

    1. If requireUserVerification is true and the authenticator cannot perform user verification, return an error code equivalent to "ConstraintError" and terminate the operation.
    2. If allowCredentialDescriptorList was not supplied, set it to a list of all credentials stored for rpId (as determined by an exact match of rpId).

2. Remove any items from allowCredentialDescriptorList that do not match a credential bound to this authenticator. A match occurs if a credential matches rpId and an allowCredentialDescriptorList item's id and type members.

3. If allowCredentialDescriptorList is now empty, return an error code equivalent to "NotAllowedError" and terminate the operation.

4. Let selectedCredential be a credential as follows. If the size of allowCredentialDescriptorList is greater than 1, return an error code equivalent to "NotAllowedError" and terminate the operation.

5. If allowCredentialDescriptorList is now empty, return an error code equivalent to "NotAllowedError" and terminate the operation.

6. Let selectedCredential be the credential matching allowCredentialDescriptorList[0].

7. Obtain user consent for using selectedCredential: 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. The prompt SHOULD display the rpId and any additional displayable data associated with selectedCredential, if possible.

8. If requireUserVerification is true, the method of obtaining user consent **must** include user verification.

9. If requireUserPresence is true, the method of obtaining user consent **must** include a test of user presence.

10. If the user denies consent or if user verification fails, return an error code equivalent to "NotAllowedError" and terminate the operation.

11. Increment the RP ID-associated signature counter or the global signature counter, depending on which approach is implemented by the authenticator, by some positive value.

12. Let authenticatorData be the byte array specified in 6.1. authenticatorData includes processedExtensions, if any, as the extensions and excluding attestedCredentialData.

13. Let signature be the assertion signature of the concatenation authenticatorData || hash using the privateKey of the selectedCredential as shown in Figure 2, below. A simple, undelimited concatenation is safe to use here because the authenticator data describes its own length. The hash of the serialized client data (which potentially has a variable length) is always the last element.

14. If requireUserVerification is true, the method of obtaining user consent **must** include user verification.

15. If requireUserPresence is true, the method of obtaining user consent **must** include a test of user presence.

16. If the user does not consent, return an error code equivalent to "NotAllowedError" and terminate the operation.

17. If any error occurred while generating the assertion signature, return an error code equivalent to "UnknownError" and terminate the operation.

18. If the user agent uses a credential associated with selectedCredential, if possible.

19. Return the user agent:

   * selectedCredential is a credential ID, if either a list of credentials of size 2 or greater was supplied by the client, or no such list was supplied. Otherwise, return only the below.

   1. Generate an assertion signature Generating an assertion
   2. Include.
   3. If any error occurred while generating the assertion signature, return an error code equivalent to "UnknownError" and terminate the operation.
   4. If the user agent uses a credential associated with selectedCredential, if possible.
   5. Return to the user agent: Note: Before performing this operation, all other operations in progress in the authenticator session must be aborted by running the authenticatorCancel operation.
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.

6.2.3. 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.

6.3. Attestation

Authenticators must also provide some form of attestation. The basic requirement is that the authenticator can produce, for each credential private key, an attestation statement verifiable by the Relying Party. Typically, this attestation statement contains a signature by an attestation private key over the attested credential public key and a provenance information for the attestation public key, enabling the Relying Party to make a trust decision. 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 authenticators any time a new public key credential is generated, in the overall form of an attestation object. The relationship of the attestation object with authenticator data (containing attested credential data) and the attestation statement is illustrated in figure 3, below.

Attestation object layout illustrating the included authenticator data (containing attested credential data) and the attestation statement.

Attestation object layout including the included authenticator data (attestated credential data) and the attestation statement.

This figure illustrates only the packed attestation statement format. Several additional attestation statement formats are defined in 8 Defined Attestation Statement Formats.

An important component of the attestation object is the attestation statement. This is a specific type of signed data object, containing statements about a public key 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 credential private key). In order to correctly interpret an attestation statement, a Relying Party needs to understand these two aspects of 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.

2. The user handle associated with selectedCredential, + authenticatorData + signature + selectedCredential.userHandle

Note: The returned userHandle value may be null, see: userHandleResult.

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.

6.2.4. 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.

6.3. Attestation

Authenticators MUST also provide some form of attestation. The basic requirement is that the authenticator can produce, for each credential public key, an attestation statement verifiable by the Relying Party. Typically, this attestation statement contains a signature by an attestation private key over the attested credential public key and a challenge, as well as a certificate or similar data providing provenance information for the attestation public key, enabling the Relying Party to make a trust decision. 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 authenticators any time a new public key credential is generated, in the overall form of an attestation object. The relationship of the attestation object with authenticator data (containing attested credential data) and the attestation statement is illustrated in figure 3, below.

Attestation Object Layout diagram Attestation object layout illustrating the included authenticator data (containing attested credential data) and the attestation statement.

This figure illustrates only the packed attestation statement format. Several additional attestation statement formats are defined in 8 Defined Attestation Statement Formats.

An important component of the attestation object is the attestation statement. This is a specific type of signed data object, containing statements about a public key 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 credential private key). In order to correctly interpret an attestation statement, a Relying Party needs to understand these two aspects of 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 TPMs and the Android OS) have previously defined attestation statement formats. This specification supports a variety of such formats in an extensible way, as defined in 6.3.2 Attestation Statement Formats.

2. The attestation type or the semantics of attestation statements and their underlying trust models. Specifically, it defines how a Relying 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 6.3.3 Attestation Types.

In general, there is no simple mapping between attestation statement formats and attestation types. For the "packed" attestation statement format defined in 8.2 Packed Attestation Statement Format, it can be used in conjunction with all attestation types, whereas other formats and types have more limited applicability.

The privacy, security, and operational characteristics of attestation depend on:

1. The attestation type, which determines the trust model,
2. The attestation statement format, which may constrain the strength of the attestation by limiting what can be expressed in an attestation statement, and
3. 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.

6.3.1. Attested credential data

Attested credential data is a variable-length byte array added to the authenticator data when generating an attestation object for a given credential. It has the following format:

Name Length (in bytes) Description
aatuid 16 The AAGUID of the authenticator.
credentialIdLength 2 Byte length L of Credential ID
credentialPublicKey ACOSE_Key encoded in
credentialPublicKeyVariable The credential public key encoded in COSE_Key format, as defined in Section 7 of [RFC8152]. The encoded credential public key MUST contain the "alg" parameter and MUST NOT contain any other optional parameters. The "alg" parameter MUST contain a COSEAlgorithmIdentifier value.

NOTE: The names in the Name column in the above table are only for reference within this document, and are not present in the actual representation of the attested credential data.

In other words, this defines the syntax of the statement. Various existing devices and platforms (such as TPMs and the Android OS) have previously defined attestation statement formats. This specification supports a variety of such formats in an extensible way, as defined in 6.3.2 Attestation Statement Formats.

2. The attestation type or the semantics of attestation statements and their underlying trust models. Specifically, it defines how a Relying 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 6.3.3 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 8.2 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 depend on:

1. The attestation type, which determines the trust model,
2. The attestation statement format, which may constrain the strength of the attestation by limiting what can be expressed in an attestation statement, and
3. 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.

6.3.1. Attested credential data

Attested credential data is a variable-length byte array added to the authenticator data when generating an attestation object for a given credential. It has the following format:

Name Length (in bytes) Description
aatuid 16 The AAGUID of the authenticator.
credentialIdLength 2 Byte length L of Credential ID
credentialPublicKey ACOSE_Key encoded in
credentialPublicKeyVariable The credential public key encoded in COSE_Key format, as defined in Section 7 of [RFC8152]. The encoded credential public key MUST contain the "alg" parameter and MUST NOT contain any other optional parameters. The "alg" parameter MUST contain a COSEAlgorithmIdentifier value.

NOTE: The names in the Name column in the above table are only for reference within this document, and are not present in the actual representation of the attested credential data.

6.3.1.1. Examples of credentialPublicKey Values encoded in COSE_Key format

This section provides examples of COSE Key-encoded Elliptic Curve and RSA public keys for the ES256, PS256, and RS256 signature algorithms. These examples adhere to the rules defined above in COSEKeyId format, and are presented in [CCDL] for clarity.

[RFC8152] Section 7 defines the general framework for all
6.3.2. 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 MUST be defined using the following template:

- Attestation statement format identifier:

* Supported attestation types:

* Syntax: The syntax of an attestation statement produced in this format, defined using [CDDL] for the extension point AttestStmtFormat

6.3.2. 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 MUST be defined using the following template:

* Attestation statement format identifier:

* Supported attestation types:

* Syntax: The syntax of an attestation statement produced in this format, defined using [CDDL] for the extension point AttestStmtFormat
ECDAA-Issuer (see [FIDOEcdaaAlgorithm]).
supports DAA using elliptic curve cryptography and bilinear
attestation (DAA) credentials from a single DAA-Issuer. These
Elliptic Curve based Direct Anonymous Attestation (ECDAA)
certificates. The attestation certificate requested most
individually. Attestation keys can be requested for each public key credential
Attestation keys can be requested for each public key credential
(which is a global correlation handle) to Privacy CA(s).
the Authenticator can limit the exposure of the endorsement key
to issue an attestation certificate for it. Using this approach,
a trusted third party, the Privacy CA. The Authenticator can
(endorsement) key. This key is used to securely communicate with
Privacy CA
meaningful protection measures for an attestation private key
specific attestation key. Instead it uses the credential private
attestation [UAFProtocol], the Authenticator does not have any
Self Attestation
In the case of self attestation, also known as surrogate basic
attestation [UAFProtocol], the Authenticator does not have any
specific attestation key. Instead it uses the credential private key 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
create multiple attestation key pairs in order to protect each Privacy CA
issue an attestation certificate for it. Using an approach like this,
the Authenticator can limit the exposure of the endorsement key
which is a global correlation handle. WebAuthn requests an Attestation CA
issue an AAIK certificate for each public key credential
individually. Attestation keys can be requested for each public key credential
Self Attestation
In the case of self attestation, also known as surrogate basic
attestation [UAFProtocol], the Authenticator does not have any
specific attestation key. Instead it uses the credential private key 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
create multiple attestation key pairs in order to protect each Privacy CA
issue an attestation certificate for it. Using an approach like this,
the Authenticator can limit the exposure of the endorsement key
which is a global correlation handle. WebAuthn requests an Attestation CA
issue an AAIK certificate for each public key credential
individually. Attestation keys can be requested for each public key credential
Self Attestation
In the case of self attestation, also known as surrogate basic
attestation [UAFProtocol], the Authenticator does not have any
specific attestation key. Instead it uses the credential private key 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
create multiple attestation key pairs in order to protect each Privacy CA
issue an attestation certificate for it. Using an approach like this,
the Authenticator can limit the exposure of the endorsement key
which is a global correlation handle. WebAuthn requests an Attestation CA
issue an AAIK certificate for each public key credential
individually. Attestation keys can be requested for each public key credential
Self Attestation
In the case of self attestation, also known as surrogate basic
attestation [UAFProtocol], the Authenticator does not have any
specific attestation key. Instead it uses the credential private key to create the attestation signature. Authenticators without
meaningful protection measures for an attestation private key
typically use this attestation type.
6.3.4. Generating an Attestation Object

To generate an attestation object (see: Figure 3) given:

attestationFormat

An attestation statement format.

authData

A byte array containing authenticator data.

hash

The hash of the serialized client data.

the authenticator MUST:

1. Let attStmt be the result of running attestationFormat's signing procedure given authData and hash.

2. Let fmt be attestationFormat's attestation statement format.

3. Return the attestation object as a CBOR map with the following syntax, filled in with variables initialized by this algorithm:

attObj = {
  authData: bytes,
  $$attStmtType
}

attStmtTemplate = {
  fmt: text,
  attStmt: { tstr => any } ; Map is filled in by each concrete attStmtType
}

; Every attestation statement format must have the above fields

attStmtTemplate .within $$attStmtType

6.3.5. Security Considerations

6.3.5.1. 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 keys (called Basic Attestation). This will anonymize the user at the risk of not being able to revoke a particular attestation key.

- A WebAuthn authenticator may be capable of dynamically generating different attestation keys and 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 related certificates.

- A WebAuthn Authenticator can implement Elliptic Curve based direct anonymous attestation (see [FIDOEcdaaAlgorithm]). Using this scheme, the authenticator generates a blinded attestation signature. This allows the Relying Party to verify the signature using the ECDAA-Issuer public key, but the attestation signature does not serve as a global correlation handle.

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

A 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 this 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-registers (or marks 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 register each intermediate attestation CA certificate 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 3.6 in [FIDOEcdaaAlgorithm]). For example, the FIDO Metadata Service (FIDOMetadataService) provides one way to access such information.

6.3.5.3. 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 facilitate isolating problems with a specific version of a device.

If the attestation root certificate is not dedicated to a single WebAuthn Authenticator device line (i.e., AAGUID), the AAGUID should be specified in the attestation certificate itself, so that it can be verified against the authenticator data.

* For COSEAlgorithmIdentifier -7 (ES256), and other ECDSA-based algorithms, a signature value is encoded as an ASN.1 DER Ecdsa-Sig-Value, as defined in [RFC3279] section 2.2.3.

Example:

```
| 3d 46 28 7b 8c 6c 8c 8c 26 1c 1b 88 f2 73 b0 9a |
| 32 a6 cf 28 09 d3 a7 8f 26 37 00 81 5a |
| 02 20 |
| 17 6c 92 bb 8e 36 c0 41 98 a2 7b 90 9b 6e 8f 13 |
```

Note: As CTAP1/U2F devices are already producing signatures values in this format, CTAP2 devices will also produce signatures values in the same format, for consistency reasons. It is recommended that any new attestation formats defined not use ASN.1 encodings, but instead represent signatures as equivalent fixed-length byte arrays without internal structure, using the same representations as used by COSE signatures as defined in [RFC8152] and [RFC8230].

* For COSEAlgorithmIdentifier -257 (RS256), sig contains the signature generated using the RSASSA-PKCS1-v1_5 signature scheme defined in section 8.2.1 in [RFC8017] with SHA-256 as the hash function. The signature is not ASN.1 wrapped.

* For COSEAlgorithmIdentifier -37 (PS256), sig contains the signature generated using the RSASSA-PSS signature scheme defined in section 8.1.1 in [RFC8017] with SHA-256 as the hash function. The signature is not ASN.1 wrapped.

7. Relying Party Operations

Upon successful execution of create() or get(), the Relying Party's script receives a PublicKeyCredential containing an
When registering a new credential, represented by an AuthenticatorAttestationResponse structure and an AuthenticationExtensionsClientOutputs structure clientExtensionResults, as part of a registration ceremony, a Relying Party MUST proceed as follows:

1. Perform JSON deserialization on the clientDataJSON field of the AuthenticatorAttestationResponse object to extract the client data C claimed as collected during the credential creation.
2. Verify that the origin in C matches the challenge that was sent to the authenticator in the create() call.
3. Verify that the challenge in C matches the challenge that was sent to the authenticator in the create() call.
4. Verify that the origin in C matches the Relying Party's origin.
5. Verify that the clientExtensions in C is a subset of the extensions values is maintained in the IANA registry of the same name [WebAuthn-Registries].
6. Verify that the origin in C matches the Relying Party's origin.
7. Verify that the type in C is the string webauthn.create.
8. Perform CBOR decoding on the attestationObject field of the AuthenticatorAttestationResponse structure to obtain the attestation statement format fmt, the authenticator data authData, and the attestation statement attStmt.
9. Verify that the value of C.type is webauthn.create.
10. Perform CBOR decoding on the tokenBinding field of the attestationObject to obtain the Token Binding for the TLS connection over which the assertion was obtained.

7.1. Registering a new credential

When registering a new credential, represented by an AuthenticatorAttestationResponse structure and an AuthenticationExtensionsClientOutputs structure clientExtensionResults, as part of a registration ceremony, a Relying Party MUST proceed as follows:

1. Let JSONtext be the result of running UTF-8 decode on the value of response.clientDataJSON.
2. Note: Using any implementation of UTF-8 decode is acceptable as long as it yields the same result as that yielded by the UTF-8 decode algorithm. In particular, any leading byte order mark (BOM) byte must be stripped.
3. Let C, the client data claimed as collected during the credential creation, be the result of running an implementation-specific JSON parse on JSONtext.
4. Note: C may be an implementation-specific data structure representation, as long as C's components are referenceable, as required by this algorithm.
5. Verify that the value of C.type matches the Relying Party's origin.
6. Verify that the value of C.tokenBinding status matches the state of the clientExtensions in C.
7. Verify that the Token Binding for the TLS connection over which the assertion was obtained.
8. Let tokenBinding be the result of running an implementation-specific JSON parse on JSONtext.
9. Verify that C.tokenBinding.id matches the base64url encoding of the Token Binding ID for the connection.
10. Compute the hash of response.clientDataJSON using SHA-256.
11. Perform CBOR decoding on the attestationObject field of the AuthenticatorAttestationResponse structure to obtain the attestation statement format fmt, the authenticator data authData, and the attestation statement attStmt.
12. Verify that the value of C.type is webauthn.create.
13. Perform CBOR decoding on the tokenBinding field of the attestationObject to obtain the Token Binding for the TLS connection over which the assertion was obtained.
14. Let attStmt be the result of running an implementation-specific JSON parse on JSONtext.

7.1.1. Registering a new credential

When registering a new credential, represented by an AuthenticatorAttestationResponse structure and an AuthenticationExtensionsClientOutputs structure clientExtensionResults, as part of a registration ceremony, a Relying Party MUST proceed as follows:

1. Let JSONtext be the result of running UTF-8 decode on the value of response.clientDataJSON.
2. Note: Using any implementation of UTF-8 decode is acceptable as long as it yields the same result as that yielded by the UTF-8 decode algorithm. In particular, any leading byte order mark (BOM) byte must be stripped.
3. Let C, the client data claimed as collected during the credential creation, be the result of running an implementation-specific JSON parse on JSONtext.
4. Note: C may be an implementation-specific data structure representation, as long as C's components are referenceable, as required by this algorithm.
5. Verify that the value of C.type matches the Relying Party's origin.
6. Verify that the value of C.tokenBinding status matches the state of the clientExtensions in C.
7. Verify that the Token Binding for the TLS connection over which the assertion was obtained.
8. Let tokenBinding be the result of running an implementation-specific JSON parse on JSONtext.
9. Verify that C.tokenBinding.id matches the base64url encoding of the Token Binding ID for the connection.
10. Compute the hash of response.clientDataJSON using SHA-256.
11. Perform CBOR decoding on the attestationObject field of the AuthenticatorAttestationResponse structure to obtain the attestation statement format fmt, the authenticator data authData, and the attestation statement attStmt.
12. Verify that the value of C.type is webauthn.create.
13. Perform CBOR decoding on the tokenBinding field of the attestationObject to obtain the Token Binding for the TLS connection over which the assertion was obtained.
14. Let attStmt be the result of running an implementation-specific JSON parse on JSONtext.

7.1.1.1. Registering a new credential

When registering a new credential, represented by an AuthenticatorAttestationResponse structure and an AuthenticationExtensionsClientOutputs structure clientExtensionResults, as part of a registration ceremony, a Relying Party MUST proceed as follows:

1. Let JSONtext be the result of running UTF-8 decode on the value of response.clientDataJSON.
2. Note: Using any implementation of UTF-8 decode is acceptable as long as it yields the same result as that yielded by the UTF-8 decode algorithm. In particular, any leading byte order mark (BOM) byte must be stripped.
3. Let C, the client data claimed as collected during the credential creation, be the result of running an implementation-specific JSON parse on JSONtext.
4. Note: C may be an implementation-specific data structure representation, as long as C's components are referenceable, as required by this algorithm.
5. Verify that the value of C.type matches the Relying Party's origin.
6. Verify that the value of C.tokenBinding status matches the state of the clientExtensions in C.
7. Verify that the Token Binding for the TLS connection over which the assertion was obtained.
8. Let tokenBinding be the result of running an implementation-specific JSON parse on JSONtext.
9. Verify that C.tokenBinding.id matches the base64url encoding of the Token Binding ID for the connection.
10. Compute the hash of response.clientDataJSON using SHA-256.
11. Perform CBOR decoding on the attestationObject field of the AuthenticatorAttestationResponse structure to obtain the attestation statement format fmt, the authenticator data authData, and the attestation statement attStmt.
12. Verify that the value of C.type is webauthn.create.
13. Perform CBOR decoding on the tokenBinding field of the attestationObject to obtain the Token Binding for the TLS connection over which the assertion was obtained.
14. Let attStmt be the result of running an implementation-specific JSON parse on JSONtext.
1. Using credential's id attribute (or the corresponding rawId, if
follows: part of an authentication ceremony, the Relying Party MUST proceed as
When verifying a given PublicKeyCredential structure (credential) as
7.2. Verifying an authentication assertion
to a different user, the Relying Party SHOULD fail this ceremony, or it
check that each credential is registered to no more than one user. If
attestation certificate chain if the client did not provide this chain
above. Also, if certificates are being used, the Relying Party must
a trusted method of determining acceptable trust anchors in step 15. Verification of attestation objects requires that the Relying Party has
cryptographic proof that the public key credential has been
+ If self attestation was used, check if self attestation is
+ Otherwise, use the X.509 certificates returned by the
procedure to verify that the verification public
correctly chains up to an acceptable root certificate.
14. 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 options.user passed to create(), by
associating it with the credentialid and credentialPublicKey in the
attestedCredentialData in authData, as appropriate for the Relying
Party's system.
15. 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 credentialPublic in the
attestedCredentialData in authData, as appropriate, for the Relying
Party's system.
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.
7.2. Verifying an authentication assertion
When verifying a given PublicKeyCredential structure (credential) as
part of an authentication ceremony, the Relying PARTY MUST proceed as
1. Using credential's id attribute (or the corresponding rawId, if
318 | This is a signal that the authenticator may be less than or equal to the signature counter value stored in conjunction with credential’s id attribute.
318 |
319 | + If the signature counter value adata.signCount is nonzero, then run the following sub-step:
320 | 1. Update the stored signature counter value, associated with credential’s id attribute, to be the value of adata.signCount.
320 |
321 | 2. If the signature counter value is greater than the signature counter value stored in conjunction with credential’s id attribute, this is a signal that the authenticator may be cloned, i.e. at least two copies of the credential are present.
321 |
322 | 3. Using credential’s id attribute (or the corresponding credential public key), look up the corresponding credential public key.
322 |
323 | 4. Let cData, aData and sig denote the value of credential’s response’s clientDataJSON, authenticatorData, and signature respectively.
323 |
324 | 5. Perform JSON deserialization on cData to extract the client data C.
324 |
325 | 6. Verify that the type in C is the string webauthn.get.
325 |
326 | 7. Verify that the clientExtensions member of C is a subset of the clientExtensions requested by the Relying Party and that the authenticatorExtensions in C is also a subset of the extensions requested by the Authenticator.
326 |
327 | 8. Verify that the rpIdHash in aData is the SHA-256 hash of the RP ID expected by the Relying Party.
327 |
328 | 9. Verify that the origin of C matches the Relying Party’s origin.
328 |
329 | 10. If user verification is required for this assertion, verify that the User Verified bit of the flags in aData is set.
329 |
330 | 11. Verify that the tokenBindingId member of C (if present) matches the Token Binding ID for the TLS connection over which the signature was sent to the authenticator in the get() call.
330 |
331 | 12. If user verification is required for this assertion, verify that the value of C.type is the string webauthn.get.
331 |
332 | 13. Verify that the rpIdHash in aData is the SHA-256 hash of the RP ID used for the signature.
332 |
333 | 14. Let hash be the result of computing a hash over the cData using the algorithm represented by the hashAlgorithm member of C.
333 |
334 | 15. Using the credential public key looked up in step 1, verify that the value of C.tokenBinding.id matches the base64url encoding of the Token Binding ID for the connection.
334 |
335 | 16. Using the credential public key looked up in step 3, verify that the User Verified bit of the flags in aData is set.
335 |
336 | 17. If the signature counter value adata.signCount is nonzero or the value stored in conjunction with credential’s id attribute is nonzero, then run the following sub-step:
336 |
337 | 1. Update the stored signature counter value, associated with credential’s id attribute, to be the value of adata.signCount.
337 |
338 | 2. If the signature counter value is greater than the signature counter value stored in conjunction with credential’s id attribute, this is a signal that the authenticator may be cloned, i.e. at least two copies of the credential are present.
338 |
339 | 3. Using credential’s id attribute (or the corresponding credential public key), look up the corresponding credential public key.
339 |
340 | 4. Let cData, aData and sig denote the value of credential’s response’s clientDataJSON, authenticatorData, and signature respectively.
340 |
341 | 5. Perform JSON deserialization on cData to extract the client data C.
341 |
342 | 6. Verify that the type in C is the string webauthn.get.
342 |
343 | 7. Verify that the clientExtensions member of C is a subset of the clientExtensions requested by the Relying Party and that the authenticatorExtensions in C is also a subset of the extensions requested by the Authenticator.
343 |
344 | 8. Verify that the rpIdHash in aData is the SHA-256 hash of the RP ID expected by the Relying Party.
344 |
345 | 9. Verify that the origin of C matches the Relying Party’s origin.
345 |
346 | 10. If user verification is required for this assertion, verify that the User Verified bit of the flags in aData is set.
346 |
347 | 11. Verify that the tokenBindingId member of C (if present) matches the Token Binding ID for the TLS connection over which the signature was sent to the authenticator in the get() call.
347 |
348 | 12. If user verification is required for this assertion, verify that the value of C.type is the string webauthn.get.
348 |
349 | 13. Verify that the rpIdHash in aData is the SHA-256 hash of the RP ID used for the signature.
349 |
350 | 14. Let hash be the result of computing a hash over the cData using the algorithm represented by the hashAlgorithm member of C.
350 |
351 | 15. Using the credential public key looked up in step 1, verify that the value of C.tokenBinding.id matches the base64url encoding of the Token Binding ID for the connection.
351 |
352 | 16. Using the credential public key looked up in step 3, verify that the User Verified bit of the flags in aData is set.
352 |
353 | 17. If the signature counter value adata.signCount is nonzero or the value stored in conjunction with credential’s id attribute is nonzero, then run the following sub-step:
353 |
354 | 1. Update the stored signature counter value, associated with credential’s id attribute, to be the value of adata.signCount.
354 |
355 | 2. If the signature counter value is greater than the signature counter value stored in conjunction with credential’s id attribute, this is a signal that the authenticator may be cloned, i.e. at least two copies of the credential are present.
355 |
356 | 3. Using credential’s id attribute (or the corresponding credential public key), look up the corresponding credential public key.
356 |
357 | 4. Let cData, aData and sig denote the value of credential’s response’s clientDataJSON, authenticatorData, and signature respectively.
357 |
358 | 5. Perform JSON deserialization on cData to extract the client data C.
358 |
359 | 6. Verify that the type in C is the string webauthn.get.
359 |
360 | 7. Verify that the clientExtensions member of C is a subset of the clientExtensions requested by the Relying Party and that the authenticatorExtensions in C is also a subset of the extensions requested by the Authenticator.
360 |
361 | 8. Verify that the rpIdHash in aData is the SHA-256 hash of the RP ID expected by the Relying Party.
361 |
362 | 9. Verify that the origin of C matches the Relying Party’s origin.
362 |
363 | 10. If user verification is required for this assertion, verify that the User Verified bit of the flags in aData is set.
363 |
364 | 11. Verify that the tokenBindingId member of C (if present) matches the Token Binding ID for the TLS connection over which the signature was sent to the authenticator in the get() call.
364 |
365 | 12. If user verification is required for this assertion, verify that the value of C.type is the string webauthn.get.
365 |
366 | 13. Verify that the rpIdHash in aData is the SHA-256 hash of the RP ID used for the signature.
366 |
367 | 14. Let hash be the result of computing a hash over the cData using the algorithm represented by the hashAlgorithm member of C.
367 |
368 | 15. Using the credential public key looked up in step 1, verify that the value of C.tokenBinding.id matches the base64url encoding of the Token Binding ID for the connection.
368 |
369 | 16. Using the credential public key looked up in step 3, verify that the User Verified bit of the flags in aData is set.
369 |
370 | 17. If the signature counter value adata.signCount is nonzero or the value stored in conjunction with credential’s id attribute is nonzero, then run the following sub-step:
370 |
private key may exist and are being used in parallel, Relying Parties should incorporate this information into their risk scoring. Whether the Relying Party updates the stored signature counter value in this case, or not, or fails the authentication ceremony or not, is Relying Party-specific.

13. If all the above steps are successful, continue with the authentication ceremony as appropriate. Otherwise, fail the authentication ceremony.

8. Defined Attestation Statement Formats

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

8.1. Attestation Statement Format Identifiers

Attestation statement formats are identified by a string, called an attestation statement format identifier, chosen by the author of the attestation statement format.

Attestation statement format identifiers SHOULD be registered per [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 the developer, in order to assure uniqueness of the identifier. All attestation statement format 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 %x5c.

All registered attestation statement format identifiers are unique amongst themselves as a matter of course.

Attestation statement formats are identified by a string, called an attestation statement format identifier, chosen by the author of the attestation statement format.

Attestation statement format identifiers SHOULD be registered per [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 the developer, in order to assure uniqueness of the identifier. All attestation statement format 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 %x5c.

Note: This means attestation statement format identifiers based on domain names MUST incorporate only LDH Labels [RFC5890].

Implementations MUST match WebAuthn attestation statement format identifiers in a case-sensitive fashion.

Attestation statement formats 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. The up-to-date list of registered WebAuthn Extensions is maintained in the IANA “WebAuthn Attestation Statement Format Identifier” registry established by [WebAuthn-Registries].

8.2. 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:

```xml
<sattStmtType>
  !<str>packed</str>
</sattStmtType>
```
the other fields.

} //

//
//
//
//

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

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

3. If ECDSA is in use, the authenticator produces sig by
concatenating authenticatorData and clientDataHash, and
signing the result using an ECDAA-Issuer public key.

The semantics of the fields are as follows:

`packedStmtFormat = {
  fmt: "packed",
  attStmt: packedStmtFormat
}`

`packedStmtFormat = {
  alg: COSEAlgorithmIdentifier,
  sig: bytes,
  x5c: [ attestCert: bytes, *(caCert: bytes) ]
}

for EDS12)

sig: bytes,
ecdaaKeyId: bytes

The semantics of the fields are as follows:

`alg: COSEAlgorithmIdentifier containing the identifier of the
algorithm used to generate the attestation signature.

sig: A byte string containing the attestation signature.

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.

ecdaaKeyId: The identifier of the ECDAA-Issuer public key. This is the
BigNumberToB encoding of the component "c" of the
ECDSA-Issuer public key as defined section 3.3. step 3.5
in [FIDOEcdaaAlgorithm].

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

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

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

3. If ECDSA is in use, the authenticator produces sig by
concatenating authenticatorData and clientDataHash, and
signing the result using an ECDAA-Issuer public key.

4. If self attestation is in use, the authenticator produces sig by
concatenating 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.

55/126
Verification procedure

Verification procedure inputs attStmt, authenticatorData and clientDataHash, the verification procedure is as follows:

1. Verify that attStmt is valid CBOR conforming to the syntax defined above and perform CBOR decoding on it to extract the contained fields.

2. If x5c is present, this indicates that the attestation type is not ECDAA. In this case:
   - o Verify that x5c is a valid signature over the concatenation of authenticatorData and clientDataHash using the attestation public key in x5c with the algorithm specified in alg.
   - o Verify that x5c meets the requirements in 8.2.1 Packed attestation statement certificate requirements.
   - o If x5c contains an extension with OID 1.3.6.1.4.5724.1.4 (id-fido-gen-ce-aaguid) verify that the value of this extension matches the aaguid in authenticatorData.
   - o If successful, return attestation type Basic and attestation trust path x5c.

3. If ecdaaKeyId is present, then the attestation type is ECDAA. In this case:
   - o Verify that sig is a valid signature over the concatenation of authenticatorData and clientDataHash using ECDAAVerify with ECDAA-Issuer public key identified by ecdaaKeyId (see [FIDO-EcdaaAlgorithm]).
   - o If successful, return attestation type ECDAA and attestation trust path ecdaaKeyId.

4. If neither x5c nor ecdaaKeyId is present, self attestation is in use:
   - o Validate that alg matches the algorithm of the credentialPublicKey in authenticatorData.
   - o Verify that sig is a valid signature over the concatenation of authenticatorData and clientDataHash using the credential public key with alg.
   - o If successful, return attestation type Self and empty attestation trust path.

8.2.1. 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:

  Subject-C

  Country where the Authenticator vendor is incorporated

  Subject-O

  Legal name of the Authenticator vendor

  Subject-CN

  Authenticator Attestation

  No stipulation.

* If the related attestation root certificate is used for multiple authenticator models, the Extension OID 1.3.6.1.4.5724.1.4 (id-fido-gen-ce-aaguid) MUST be present, containing the AAGUID as value.

Note that an X.509 Extension encodes the DER-encoding of the value in an OCTET STRING. Thus, the AAGUID must be wrapped in two OCTET STRINGS to be valid. Here is a sample, encoded Extension structure:

04 12                                   -- OCTET STRING

This is as follows:

str = "Legal name of the Authenticator vendor (UTF8String)"
ECDAAVerify (04 12, alg) returns (true, str)
ECDAAVerify (04 12, alg) returns (true, str)
The attestation signature, in the form of a TPMT_SIGNATURE

BigNumberToB encoding of the component "c" as defined in section 3.3, step 3.5 in [FIDOEcdaaAlgorithm].

The identifier of the ECDAA-Issuer public key. This is the x5c certificate chain, in X.509 encoding.

A COSEAlgorithmIdentifier containing the identifier of the algorithm used to generate the attestation signature.

The version of the TPM specification to which the signature conforms.

The AIK certificate used for the attestation and its certificate chain, in X.509 encoding.

The identifier of the ECDAA-Issuer public key. This is the ecdaaKeyId.

The attestation signature, in the form of a TPMT_SIGNATURE structure as specified in [TPMv2-Part2] section 11.3.4.
The TPMS_ATTEST structure over which the above signature was computed, as specified in [TPMv2-Part2] section 10.12.8.

pubArea

The TPMT_PUBLIC structure (see [TPMv2-Part2] section 12.2.4) used by the TPM to represent the credential public key.

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.

Generate a signature using the procedure specified in [TPMv2-Part2] Section 18.2, using the private key and setting the extraData parameter to the digest of attToBeSigned using the hash algorithm corresponding to the "alg" signature algorithm. (For the "RS256" algorithm, this would be a SHA-256 digest.)

Set the pubArea field to the public area of the credential unique fields of pubArea is identical to the credentialPublicKey defined above, and perform CBOR decoding on it to extract the contained fields.

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 the hash of attToBeSigned using the hash algorithm employed in "alg".
+ Verify that attested contains a TPM_CERTIFY_INFO structure, whose name field contains a valid Name for pubArea, as computed using the algorithm in the nameAlg field of pubArea, using the procedure specified in [TPMv2-Part1] section 16.

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

+ Verify that 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 8.3.1 TPM attestation statement certificate requirements.
Let authenticatorData denote the authenticator data for the
Signing procedure
alg: COSEAlgorithmIdentifier,
X.509 certificates. See the Android developer documentation. Its
An Android key attestation statement consists simply of the
Basic Attestation
Attestation statement format identifier
android-key
Attestation types supported
Basic Attestation
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:

SattStmtType /= (  
  fmt: "android-key",  
  attStmt: androidStmtFormat  
)

androidStmtFormat = (  
  aq: COSEAlgorithmIdentifier,  
  sig: bytes,  
  x5c: [ credCert: bytes, *(caCert: bytes) ]  
)

Signing procedure
Let authenticatorData denote the authenticator data for the
attestation, and let clientDataHash denote the hash of the

8.5. 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. This attestation does not provide information regarding provenance of the authenticator and its associated data. Therefore platform-provided authenticators should make use of the Android Key Attestation when available, even if the SafetyNet API is also present.

Attestation statement format identifier

```
android-safetynet
```

Attestation types supported

```
Basic Attestation
```

Syntax

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

```attStmt: safetynetStmtFormat
<<<attStmtType //= (android-safetynet) providing clientDataHash as the challenge value (e.g., by using setAttestationChallenge).
set x5c to the returned value.
```

The authenticator produces sig by concatenating authenticatorData and clientDataHash, and signing the result using the credential private key. It sets alg to the algorithm of the signature format.

Verification procedure

Given the verification procedure inputs attStmt, authenticatorData and clientDataHash, the verification procedure is as follows:

+ Verify that attStmt is valid CBOR conforming to the syntax defined above, and perform CBOR decoding on it to extract the contained fields.
+ Verify that the public key in the first certificate in the series of certificates represented by the signature matches the credentialPublicKey in the attestedCredentialData in authenticatorData.
+ Verify that in the attestation certificate extension data:
  o The value of the attestationChallenge field is identical to clientDataHash.
  o The AuthorizationList.allApplications field is not present, since PublicKeyCredentials must be bound to the RP ID.
  o The value in the AuthorizationList.origin field is equal to KM_TAG_GENERATED.
  o The value in the AuthorizationList.purpose field is equal to KM_PURPOSE_SIGN.
+ If successful, return attestation type Basic with the attestation trust path set to the entire attestation statement.

## 8.5. 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. This attestation does not provide information regarding provenance of the authenticator and its associated data. Therefore platform-provided authenticators should make use of the Android Key Attestation when available, even if the SafetyNet API is also present.

Attestation statement format identifier

```
android-safetynet
```

Attestation types supported

```
Basic Attestation
```

Syntax

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

+ If successful, return attestation type Basic with the attestation trust path set to x5c.
The syntax of a FIDO U2F attestation statement is defined as follows:

```javascript
attStmtType //= (
  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 UTF-8 encoded result of the getJwsResult() call of the SafetyNet API. This value is a JWS [RFC7515] object (see SafetyNet online documentation) in Compact Serialization.

### 8.6. FIDO U2F Attestation Statement Format

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

```javascript
attStmt: u2fStmtFormat
```

**Attestation types supported**

- Basic Attestation, Self Attestation, Privacy CA

**Syntax**

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

```javascript
$sattStmtType //= (  fmt: "fido-u2f",
  attStmt: u2fStmtFormat
)
```
u2fStmtFormat = { 
  xSc: [ attestnCert: bytes, * (caCert: bytes) ], 
  sig: bytes
}

The semantics of the above fields are as follows:

- xSc: 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. The signature was calculated over the (raw) U2F registration response message [FIDO-U2F-Message-Formats] received by the platform from the authenticator.

**Signing procedure**

1. Verify that attStmt is valid CBOR conforming to the syntax defined above, and perform CBOR decoding on it to extract the contained fields.

2. Let attCert be the value of the first element of xSc. Let certificate public key be the public key conveyed by attCert.

3. Extract the claimed clientDataHash from authenticatorData, and the claimed credentialId and credentialPublicKey from authenticatorData.attestedCredentialData.

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

Generate a Registration Response Message as specified in [FIDO-U2F-Message-Formats] section 4.3, with the application parameter set to the SHA-256 hash of the RP ID associated with the given credential, the challenge parameter set to tbsHash, and the key handle parameter set to the credential ID of the given credential. Set the raw signature part of this Registration Response Message (i.e., without the use public key, key handle, and attestation certificate fields) as sig and set the attestation certificates of the attestation public key as xSc.

**Verification procedure**

Given the verification procedure inputs attStmt, authenticatorData and clientDataHash, the verification procedure is as follows:

1. Verify that attStmt is valid CBOR conforming to the syntax defined above and perform CBOR decoding on it to extract the contained fields.

2. Let attCert be value of the first element of xSc. Let certificate public key be the public key conveyed by attCert.

3. If certificate public key is not an Elliptic Curve (EC) public key over the P-256 curve, terminate this algorithm and return an appropriate error.

4. Extract the claimed clientDataHash from authenticatorData, and the claimed credentialId and credentialPublicKey from authenticatorData.attestedCredentialData.

5. Convert the COSEHasKey formatted credentialPublicKey (see Section 7 of [RFC8152]) to CTAP1/U2F public Key format [FIDO-CTAP].

   - Let publicKeyU2F represent the result of the conversion operation and set its first byte to 0x04. Note: This signifies uncompressed ECC key format.
   - Extract the value corresponding to the "-2" key (representing x coordinate from credentialPublicKey). If size differs or "-2" key is not found, terminate this algorithm and return an appropriate error.
   - Confirm its size to be of 32 bytes and concatenate it with publicKeyU2F. If size differs or "-2" key is not found, terminate this algorithm and return an appropriate error.
9. WebAuthn Extensions

The mechanism for generating public key credentials, as well as requesting and generating Authentication assertions, as defined in 5 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.credentials.create() extension request parameters and response values for registration extensions.
- *navigator.credentials.get() extension request parameters and response values for authentication extensions.
- * Client extension processing for registration extensions and authentication extensions.

When creating a public key credential or requesting an authentication assertion, a Relying Party can request the use of a set of extensions. These extensions will be invoked during the requested operation if they are supported by the client and/or the authenticator. The Relying Party sends the client extension input for each extension in the get() call (for authentication extensions) or create() call (for registration extensions) to the client platform. The client platform performs client

8. If successful, return attestation type Basic with the attestation trust path set to x5c.

5. Let verificationData be the concatenation of (0x00 || rpIdHash || clientDataHash || credentialId || publicKeyU2F) (see Section 4.3 of [FIDO-U2F-Message-Formats]).

4. Verify the sig using verificationData and certificate public key per [SEC1].

3. If successful, return attestation trust path set to x5c.

2. If successful, return attestation type Basic with the attestation trust path set to x5c.

1. Extract the value corresponding to the "-3" key (representing y coordinate) from credentialPublicKey, confirm its size to be of 32 bytes and concatenate it with publicKeyU2F. If size differs or "-3" key is not found, terminate this algorithm and return an appropriate error.

8.7. None Attestation Statement Format

The none attestation statement format is used to replace any authenticator-provided attestation statement when a Relying Party indicates it does not wish to receive attestation information, see 5.4.6 Attestation Conveyance Preference enumeration (enum AttestationConveyancePreference).

The syntax of a none attestation statement is defined as follows:

```plaintext
$attStmtType //= none
fmt: "none":
attStmt: emptyMap
```

emptyMap = {}  

Signin procedure  
Return the fixed attestation statement defined above.

Verification procedure  
Return attestation type None with an empty trust path.

9. WebAuthn Extensions

The mechanism for generating public key credentials, as well as requesting and generating Authentication assertions, as defined in 5 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.credentials.create() extension request parameters and response values for registration extensions.
- *navigator.credentials.get() extension request parameters and response values for authentication extensions.
- * Client extension processing for registration extensions and authentication extensions.

When creating a public key credential or requesting an authentication assertion, a Relying Party can request the use of a set of extensions. These extensions will be invoked during the requested operation if they are supported by the client and/or the authenticator. The Relying Party sends the client extension input for each extension in the get() call (for authentication extensions) or create() call (for registration extensions) to the client platform. The client platform performs 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 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 assertion 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 corresponding call (for registration extensions) or the get() call (for authenticator 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 authenticator extension input value as the value. The authenticator, in turn, performs additional processing for the extensions that it supports, and returns the CBOR authenticator extension output value for each specified by the extension. Part of the processing for authenticator extensions is to use the authenticator extension output value 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 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 value by simply encoding the client extension input in CBOR. All WebAuthn extensions MUST be defined in such a way that this approach does not endanger the user's security or privacy. For instance, if an extension requires client processing, it could be defined in such a way that it ensures such a natively 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 it does not understand by encoding the authenticator extension output value into JSON, provided that the CBOR output uses only types present in JSON.

When clients choose to pass through extensions they do not recognize, the JavaScript values in the client extension inputs are converted to CBOR values in the authenticator extension output. The JavaScript value is an %ArrayBuffer%, it is converted to a CBOR byte array. When the JavaScript value is a non-integer number, it is converted to a 64-bit CBOR floating point number. Otherwise, when the JavaScript type corresponds to a JSON type, the conversion is done using the rules defined in Section 3.2.8.5 of [RFC7904]. The exception is when the JavaScript value is a Boolean value. Then, the conversion is done using the rules defined in Section 4.2 of [RFC7049] (Converting from JSON to CBOR), but operating on inputs of JavaScript type values rather than inputs of JSON type values. Once these conversions are performed, the canonicalization of the resulting CBOR MUST be performed using the CTAP2 canonical CBOR encoding form.

Likewise, when clients receive outputs from extensions they have passed that they do not recognize, the CBOR values in the
9.1. Extension Identifiers

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

Extension identifiers SHOULD be registered per the [WebAuthn-Registrations] for Web Authentication (WebAuthn) specification. 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 %x2c. Implementations MUST match WebAuthn extension identifiers in a case-sensitive fashion.

Extensions that may exist in multiple versions should take care to not clash with their identifier. In effect, different versions are thus treated as different extensions, e.g., myCompany_extension_01

10 Defined Extensions defines an initial set of extensions and their identifiers. See the IANA "WebAuthn Extension Identifier" registry established by [WebAuthn-Registrations] for an up-to-date list of registered WebAuthn Extension Identifiers.

9.2. Defining extensions

A definition of an extension must specify an extension identifier, a client extension input argument to be sent via the get() or create() call, the client extension processing rules, and a client extension output value. If the extension communicates with the authenticator (meaning it is an authenticator extension), it must also specify the CBOR authenticator extension input argument sent via the authenticatorGetAssertion or authenticatorMakeCredential call, the authenticator extension processing rules, and the CBOR authenticator extension output value.

Any client extension that is processed by the client MUST return a client extension output value so that the Relying Party knows that the extension was understood and processed. Likewise, any authenticator extension that does not otherwise require any result values MUST return a value and SHOULD return a CBOR Boolean authenticator extension output result, set to true to signify that the extension was understood and processed.
9.3. Extending request parameters

An extension defines one or two request arguments. The client extension input, which is a value that can be encoded in JSON, is passed from the Relying Party to the client in the get() or create() call, while the CBOR authenticator extension input is passed from the client to the authenticator for authenticator extensions during the processing of these calls.

A Relying Party simultaneously requests the use of an extension and sets its client extension input by including an entry in the extensions option to the create() or get() call. The entry key is the extension identifier and the value is the client extension input.

var assertionPromise = navigator.credentials.get({
  publicKey: {},

  // The challenge must be produced by the server, see the Security Considerations.
  challenge: new Uint8Array([4,99,22 /* 29 more random bytes generated by the server */)];

  extensions: { 
    "webauthnExample_fooBar": 42
  }
});

// The challenge must be produced by the server, see the Security Considerations.
var assertionPromise = navigator.credentials.get({
  publicKey: {},

  // The challenge must be produced by the server, see the Security Considerations.
  challenge: new Uint8Array([4,99,22 /* 29 more random bytes generated by the server */]);

  extensions: { 
    "webauthnExample_fooBar": 42
  }
});

Extension definitions MUST specify the valid values for their client extension input. Clients SHOULD ignore extensions with an invalid client extension input. If an extension does not require any parameters from the Relying Party, it SHOULD be defined as taking a Boolean 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 extension input. Extensions that have a processing MUST specify the method of computing the authenticator extension input from the client extension input. For extensions that do not require any input parameters and are defined as taking a Boolean client extension input value set to true, this method SHOULD consist of passing an authenticator extension input value 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.

9.4. Client extension processing

Extensions MAY define additional processing requirements on the client platform during the creation of credentials or the assertion of an assertion. The client extension input for the extension is used an input to this client processing. Supported client extensions are recorded as a dictionary in the client data with the key clientExtensions. For each such extension, the client adds an entry to this dictionary with the extension identifier as the key, and the extension's client extension input as the value.

Likewise, the client extension outputs are represented as a dictionary in the result of getClientExtensionResults() with extension identifiers as keys, and the client extension output value of each extension as the value. Like the client extension input, the client extension output is a value that can be encoded in JSON.

Extensions that require authenticator processing MUST define the process by which the client extension input can be used to determine the CBOR authenticator extension input and the process by which the CBOR authenticator extension output can be used to determine the client extension output.
9.5. Authenticator extension processing

The CBOR authenticator extension input value of each processed authenticator extension is included in the extensions data part of the authenticator request. This part is a CBOR map, with CBOR extension identifier values as keys, and the CBOR authenticator extension input value of each extension as the value.

Likewise, the extension output is represented in the authenticator data as a CBOR map with CBOR extension identifiers as keys, and the CBOR authenticator extension output value of each extension as the value.

The authenticator extension processing rules are used create the authenticator extension output from the authenticator extension input, and possibly also other inputs, for each extension.

9.6. Example Extension

This section is not normative.

To illustrate the requirements above, consider a hypothetical registration extension and authentication extension "Geo". This extension, if supported, enables a geolocation location to be returned from the authenticator or client to the Relying Party.

The extension identifier is chosen as webauthnExample_geo. The client extension input 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:

```
var assertionPromise = navigator.credentials.get({
    // The challenge must be produced by the server, see the Security Considerations
    challenge: new Uint8Array([11,103,35,29 more random bytes generated by the server '7']),
    allowCredentials: [{ // Empty filter
        extensions: { 'webauthnExample_geo': true }
    }],

    // The extension also requires the client to set the authenticator parameter to the fixed value true,
    // and possibly also other inputs, for each extension.
    extensions: { 'webauthnExample_geo': true }
});
```

Likewise, the extension output is represented in the extensions part of the authenticator data. The extensions part of the authenticator data is a CBOR map where each key is an extension identifier and the corresponding value is the authenticator extension output for that extension.

For each supported extension, the authenticator extension processing rule for that extension is used create the authenticator extension output from the authenticator extension input and possibly also other inputs.
10. Defined Extensions

This section defines the 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.

10.1. FIDO AppId Extension (appid)

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 Relying Parties to specify an appId [FIDO-APPID] to overwrite the otherwise computed rpId. This extension is only valid if used during the get() call; other usage will result in client error.

Client extension input

A single JSON string specifying a FIDO appId.

Client extension processing

If rpId is present, return a DOMException whose name is "NotAllowedError", and terminate this algorithm (5.1.4.1)

Otherwise, replace the calculation of rpId in Step 6 of 5.1.4.1 with the following procedure: The client uses the value of appId to perform the PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method. If valid, the value of rpId for all client processing should be replaced by the value of appId.

Client extension output

Returns the JSON value true to indicate to the RP that the extension was acted upon.

10. Defined Extensions

This section defines the 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.

10.1. FIDO AppId Extension (appid)

This client extension allows Relying Parties that have previously registered a credential using the legacy FIDO JavaScript APIs to request an assertion. The FIDO APIs use an alternative identifier for relying parties called an AppId [FIDO-APPID], and any credentials created using those APIs will be bound to that identifier. Without this extension, they would need to be re-registered in order to be bound to an RP ID.

This extension does not allow FIDO-compatible credentials to be created. Thus, credentials created with WebAuthn are not backwards compatible with the FIDO JavaScript APIs.

Client extension input

A single USVString specifying a FIDO AppId.

Client extension processing

1. If present in a create() call, return a "NotSupportedError" DOMException--this extension is only valid when requesting an assertion.
2. Let facetId be the result of passing the caller's origin to the FIDO algorithm for determining the FacetId of a calling application.
3. Let appId be the extension input.
4. Pass facetId and appId to the FIDO algorithm for determining if a caller's FacetId is authorized for an AppId. If that algorithm rejects appId then return a "SecurityError" DOMException.
5. When building allowCredentialDescriptorList, if a U2F authenticator indicates that a credential is inapplicable (i.e. by returning SW WRONG_DATA) then the client MUST retry with the U2F application parameter set to the SHA-256 hash of appId. If this results in an applicable credential, the client MUST include the credential in allowCredentialDescriptorList.
6. The value of appId then replaces the rpId parameter of authenticatorGetAssertion.

Client extension output

Returns the value true to indicate to the RP that the extension was acted upon.
**10.2. Simple Transaction Authorization Extension (txAuthSimple)**

This registration extension and 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.

- **txAuthSimpleInput**:
  - A single UTF-8 decoded USVString prompt.

- **txAuthSimpleOutput**:
  - A single UTF-8 decoded USVString.

**CDDL:**

```
txAuthSimpleInput = (tstr)
```

- **txAuthSimpleProcessing**:
  - 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.

**10.3. Generic Transaction Authorization Extension (txAuthGeneric)**

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.

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

- **txAuthGenericProcessing**:
  - The authenticator **must** display the prompt to the user before performing either user verification or test of user presence.

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

**CDDL:**

```
txAuthSimpleOutput = (tstr)
```

**generic**

```
authenticator.
```

```
authenticators.
```

**CDDL:**

```
xAuthGenericInput = (tstr)
```

- **Extension identifier**
  - \(\text{txAuthGeneric}\)
423/428 Client extension input
423/428 A CBOR map defined as follows:
423/428
txAuthGenericArg = {
423/428     contentType: text,  // MIME-Type of the content, e.g., "image/png"
423/428     content: bytes
423/428 }  

423/428 Client extension processing
423/428 None, except creating the authenticator extension input from the client extension input.
423/428
423/428 Client extension output
423/428 Returns the base64url encoding of the authenticator extension output value as a JSON string

423/428 Authenticator selection input
423/428 The authenticator inputs encoded as a CBOR map.
423/428
423/428 Authenticator extension processing
423/428 The authenticator MUST display the content to the user before performing either user verification or 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.
423/428
423/428 Authenticator extension output
423/428 The hash value of the content which was displayed. The authenticator MAY use the same hash algorithm as it uses for the signature itself.

10.4. 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 tightly control the experience around credential creation.

Extension identifier
authnSel

Client extension input
authnSel:

typedef sequence<AAGUID> AuthenticatorSelectionList;

427/428 Each AAGUID corresponds to an authenticator model that is acceptable to the Relying Party for this credential creation.
427/428 The list is ordered by decreasing preference.
427/428 An AAGUID is defined as an array containing the globally unique identifier of the authenticator model being sought.

typedef BufferSource AAGUID;

Client extension processing

429/431 dictionary txAuthGenericArg {
429/431     contentType: text,  // MIME-Type of the content, e.g., "image/png"
429/431     content: bytes
429/431 }  

430/431 partial dictionary AuthenticationExtensionsClientInputs {
430/431     txAuthGenericArg txAuthGeneric;
430/431 }  

432/433 Client extension processing
432/433 None, except creating the authenticator extension input from the client extension input.
432/433
432/433 Client extension output
432/433 Returns the authenticator extension output value as an ArrayBuffer.
432/433
433/434 partial dictionary AuthenticationExtensionsClientOutputs {
433/434     ArrayBuffer txAuthGeneric;
433/434 }  

435/436 Authenticator extension input
435/436 The client extension input encoded as a CBOR map.
435/436
435/436 Authenticator extension processing
435/436 The authenticator MUST display the content to the user before performing either user verification or 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.
435/436
435/436 Authenticator extension output
435/436 The hash value of the content which was displayed. The authenticator MUST use the same hash algorithm as it uses for the signature itself.

10.4. 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 tightly control the experience around credential creation.

Extension identifier
authnSel

Client extension input
authnSel:

typedef sequence<AAGUID> AuthenticatorSelectionList;

427/428 Each AAGUID corresponds to an authenticator model that is acceptable to the Relying Party for this credential creation.
427/428 The list is ordered by decreasing preference.
427/428 An AAGUID is defined as an array containing the globally unique identifier of the authenticator model being sought.

typedef BufferSource AAGUID;

Client extension processing
10.5. Supported Extensions Extension (exts)

This registration extension enables the Relying Party to determine which extensions the authenticator supports.

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

- **Client extension output**
  - Returns the list of supported extensions as a JSON array of extension identifier strings.

- **Authenticator extension input**
  - The Boolean value true, encoded in CBOR (major type 7, value 1).

- **Authenticator extension processing**
  - The authenticator sets the authenticator extension output to be a list of extensions that the authenticator supports, as defined below. This extension can be added to attestation objects.

- **Authenticator extension output**
  - The SupportedExtensions extension is a list (CBOR array) of extension identifier (UTF-8 encoded strings).

10.6. User Verification Index Extension (uvi)

- **Client extension output**
  - Returns the JSON value true to indicate to the RP that the extension was acted upon.

- **Authenticator extension input**
  - None.

- **Authenticator extension processing**
  - None.

- **Authenticator extension output**
  - None.

10.5. Supported Extensions Extension (exts)

This registration extension enables the Relying Party to determine which extensions the authenticator supports.

- **Client extension input**
  - The Boolean value true, encoded in CBOR (major type 7, value 1).

- **Client extension output**
  - Returns the list of supported extensions as an array of extension identifier strings.

- **Authenticator extension input**
  - The Boolean value true, encoded in CBOR (major type 7, value 1).

- **Authenticator extension processing**
  - The authenticator sets the authenticator extension output to be a list of extensions that the authenticator supports, as defined below. This extension can be added to attestation objects.

- **Authenticator extension output**
  - The SupportedExtensions extension is a list (CBOR array) of extension identifier (UTF-8 encoded strings).

71/126
This registration extension and authentication extension enables use of a user verification index.

Extension identifier
uvi

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
Returs a JSON string containing the base64url encoding of the authenticator extension output

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

Authenticator extension processing
The authenticator sets the authenticator extension output to be the 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 extension output
The user verification index (UVI) is a value uniquely identifying a user verification data record. The UVI is encoded as CBOR byte strings (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 || represents concatenation, and 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

-- [RP ID]= hash (32 bytes)
-- UP and ED set
-- (initial) signature counter
-- all public key alg etc.
-- extension: CBOR map of one elemen
-- Key 1: CBOR text string of 3 byte
10.7. 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 the 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 as defined by The W3C Geolocation API Specification.

Values represented by the "double" type in JavaScript are represented as 64-bit CBOR floating point numbers. Per the Geolocation specification, the "latitude", "longitude", and "accuracy" values are required and other values such as "altitude" are optional.

Authenticator extension input

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

Authenticator extension processing

Determine the Geolocation value.

Authenticator extension output


Values represented by the "double" type in JavaScript are represented as 64-bit CBOR floating point numbers. Per the Geolocation specification, the "latitude", "longitude", and "accuracy" values are required and other values such as "altitude" are optional.
86  -- Value 2; array of 5 elements
86  -- Element 1: CBOR text string of 8 bytes
86  -- "latitude" [UTF-8 encoded] string
86  -- "latitude" [UTF-8 encoded] string
86  -- Element 2: Latitude as CBOR encoded double-precision float
86  -- Element 3: CBOR text string of 8 bytes
86  -- "altitude" [UTF-8 encoded] string
86  -- Element 4: Longitude as CBOR encoded double-precision float
86  -- Element 5: CBOR text string of 8 bytes
86  -- "latitude" [UTF-8 encoded] string
86  -- Element 6: Altitude as CBOR encoded double-precision float

10.8. User Verification Method Extension (uvm)

This registration extension and authentication extension enables use of one or more user verification methods indicating the method(s) used by the user to authorize the operation, as defined below.

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.

matcherProtectionType
The method used by the authenticator to protect the matcher that performs user verification. Available values are defined in [FIDOReg], "Matcher Protection Types" section.

If >3 factors can be used in an authentication instance the authenticator vendor must select the 3 factors it believes will be most relevant to the Server to include in the UVM.

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

```plaintext
... -- [RP ID=] hash (32 bytes)
  81   -- UP and ED set
  00 00 00 01 -- (initial) signature counter
  00 00 00 01 -- all public key alg etc.
  83   -- extension: CBOR map of one element
    63   -- Key 1: CBOR text string of 3 bytes
    75 76 6d -- "uvm" [=UTF-8 encoded=] string
    82   -- Value 1: CBOR array of length 2 indicating two factor usage
    83   -- Item 1: CBOR array of length 3
      02   -- Subitem 1: CBOR integer for User Verification Method
      04   -- Subitem 1: CBOR integer for Matcher Protection Type TEE
      02   -- Subitem 3: CBOR short for Matcher Protection Type TEE
    83   -- Item 2: CBOR array of length 3
      04   -- Subitem 1: CBOR integer for User Verification Method
      01   -- Subitem 2: CBOR short for Key Protection Type Software
      01   -- Subitem 3: CBOR short for Matcher Protection Type Software
```

If >3 factors can be used in an authentication instance the authenticator vendor MUST select the 3 factors it believes will be most relevant to the UVM.

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

```plaintext
... -- [RP ID=] hash (32 bytes)
  81   -- UP and ED set
  00 00 00 01 -- (initial) signature counter
  00 00 00 01 -- all public key alg etc.
  A1     -- extension: CBOR map of one element
    63   -- Key 1: CBOR text string of 3 bytes
    75 76 6d -- "uvm" [=UTF-8 encoded=] string
    82   -- Value 1: CBOR array of length 2 indicating two factor usage
    83   -- Item 1: CBOR array of length 3
      02   -- Subitem 1: CBOR integer for User Verification Method
      04   -- Subitem 1: CBOR integer for Matcher Protection Type TEE
      02   -- Subitem 3: CBOR short for Matcher Protection Type TEE
      04   -- Subitem 1: CBOR integer for User Verification Method
      01   -- Subitem 2: CBOR short for Key Protection Type Software
      01   -- Subitem 3: CBOR short for Matcher Protection Type Software
```

10.9. Biometric Authenticator Performance Bounds Extension (biometricPerfBounds)

This registration extension allows Relying Parties to specify the required performance bounds for selecting biometric authenticators as candidates to be employed in a registration ceremony.

- Extension identifier: biometricPerfBounds
- Client extension input: Biometric performance bounds: dictionary authenticatorBiometricPerfBounds{
  float FAR;
  float FRR;
}

The FAR is the maximum false acceptance rate for a biometric
11. IANA Considerations

11.1. WebAuthn Attestation Statement Format Identifier Registrations

This section registers the attestation statement formats defined in Section 8 Defined Attestation Statement Formats in the IANA "WebAuthn Attestation Statement Format Identifier" registry established by [WebAuthn-Registraries].

* WebAuthn Attestation Statement Format Identifier: packed

  Description: The "packed" attestation statement format is a WebAuthn-optimized format for attestation. 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 8.2 Packed Attestation Statement 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 rawData and signature fields are computed differently.

  Specification Document: Section 8.3 TPM Attestation Statement Format of this specification

* WebAuthn Attestation Statement Format Identifier: android-key

  Description: Platform-provided authenticators based on versions "N", and later, may provide this proprietary "hardware attestation" statement.

  Specification Document: Section 8.4 Android Key Attestation Statement Format of this specification

* WebAuthn Attestation Statement Format Identifier: android-safetynet

  Description: Android-based, platform-provided authenticators may produce an attestation statement based on the Android SafetyNet API.

  Specification Document: Section 8.5 Android SafetyNet Attestation Statement Format of this specification

* WebAuthn Attestation Statement Format Identifier: fido-u2f

  Description: Used with FIDO U2F authenticators

  Specification Document: Section 8.6 FIDO U2F Attestation Statement Format of this specification

11.2. WebAuthn Extension Identifier Registrations

This section registers the extension identifier values defined in Section 9 WebAuthn Extensions in the IANA "WebAuthn Extension Identifier" registry established by [WebAuthn-Registraries].

* WebAuthn Extension Identifier: authenticated-allow-by-the-relying-party

  Description: The extension is the maximum false rejection rate for a biometric authenticator allowed by the Relying Party.


* WebAuthn Extension Identifier: client-extension-processing

  Description: This extension can only be used during create(). If the client supports this extension, it MUST NOT use a biometric authenticator whose FAR or FRR does not meet the bounds as provided. The client can obtain information about the biometric authenticator's performance from authoritative sources such as the FIDO Metadata Service ([FIDO-Metadata-Service]) (see Sec. 3.2 of [FIDO-UAFAuthenticator-Metadata-Statement]).


* WebAuthn Extension Identifier: client-extension-output

  Description: Returns the JSON true value to indicate to the RP that the extension was acted upon


* WebAuthn Extension Identifier: authenticator-extension-input

  Description: None.


* WebAuthn Extension Identifier: authenticator-extension-processing

  Description: None.


* WebAuthn Extension Identifier: authenticator-extension-output

  Description: None.

This section registers identifiers for RSASSA-PKCS1-v1_5 [RFC8017].

11.3. COSE Algorithm Registrations

This section registers identifiers for RSASSA-PKCS1-v1_5 [RFC8017] algorithms using SHA-2 and SHA-1 hash functions in the IANA COSE
4. The client platform connects to the authenticator, performing any
3. The client platform searches for and locates the authenticator.
2. The Relying Party script runs the code snippet below.
1. The user visits example.com, which serves up a script. At this
point, the user may already be logged in using a legacy username
and password, or additional authenticator, or other means
registered with the server. In this flow, the Relying Party does not
have a preference for platform authenticator or roaming authenticators.
One example of such an authenticator would be a smart phone. Other
authenticator types are also supported by this API, subject to
specific implementation considerations. Specifically, the client
platform needs to display any prompts that would otherwise be shown by
the authenticator, and the authenticator needs to allow the client
platform to enumerate all the authenticator's credentials so that the
client can have information to show appropriate prompts.

12. 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
client can have information to show appropriate prompts.

12.1. Registration

This is the first-time flow, in which a new credential is created and
registered with the server. In this flow, the Relying Party does not
have a preference for platform authenticator or roaming authenticators.

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

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.

12. 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
client can have information to show appropriate prompts.

12.1. Registration

This is the first-time flow, in which a new credential is created and
registered with the server. In this flow, the Relying Party does not
have a preference for platform authenticator or roaming authenticators.

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

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.
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

7. If a new credential was created,

8. 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.

9. 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.

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

var publicKey = {
    // The challenge must be produced by the server, see the Security Considerations.
    challenge: new Uint8Array([21, 31, 105 /* 29 more random bytes generated by the server */]),

    // Relying Party:
    rp: {
        name: "Acme",
    },

    // User:
    user: {
        id: Uint8Array.from(window.atob("MIIBkzCCATigAwIBAjCCAZMwggE4oAMCAQIwggGTMIIeQIwMII=b+DgYDQYJK4...
    },

    // This Relying Party will accept either an ES256 or RS256 credential, but
    // prefers an ES256 credential.
    pubKeyCredParams: [{
        type: "public-key",
        alg: -27 // "ES256" as registered in the IANA COSE Algorithms registry
    }, {
        type: "public-key",
        alg: -257 // Value registered by this specification for "RS256"
    }],

    // Note: The following call will cause the authenticator to display UI.
    navigator.credentials.create({ publicKey })
    // then function (newCredentialInfo) {
    // Send new credential info to server for verification and registration.
    // (function (error) {
    // No acceptable authenticator or user refused consent. Handle appropriately
    // });
    // }
};
```

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

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

var publicKey = {
    // The challenge must be produced by the server, see the Security Considerations.
    challenge: new Uint8Array([21, 31, 105 /* 29 more random bytes generated by the server */]),

    // Relying Party:
    rp: {
        name: "ACME Corporation"
    },

    // User:
    user: {
        id: Uint8Array.from(window.atob("MIIBkzCCATigAwIBAjCCAZMwggE4oAMCAQIwggGTMIIeQIwMII=b+DgYDQYJK4...
    },

    // This Relying Party will accept either an ES256 or RS256 credential, but
    // prefers an ES256 credential.
    pubKeyCredParams: [{
        type: "public-key",
        alg: -27 // "ES256" as registered in the IANA COSE Algorithms registry
    }, {
        type: "public-key",
        alg: -257 // Value registered by this specification for "RS256"
    }],

    // Note: The following call will cause the Authenticator to display UI.
    navigator.credentials.create({ publicKey })
    // then function (newCredentialInfo) {
    // Send new credential info to server for verification and registration.
    // (function (error) {
    // No acceptable authenticator or user refused consent. Handle appropriately
    // });
    // }
};
```
12.2. Registration Specifically with User Verifying Platform Authenticator

This is the flow for when the Relying Party is specifically interested in creating a public key credential with a user-verifying platform authenticator.

1. The user visits example.com and clicks on the login button, which redirects the user to login.example.com.
2. The user enters a username and password to log in. After successful login, the user is redirected back to example.com.
3. The Relying Party script runs the code snippet below.
4. The user agent asks the user whether they are willing to register with the Relying Party using an available platform authenticator.
5. If the user is not willing, terminate this flow.
6. The user is shown appropriate UI and guided in creating a credential using one of the available platform authenticators. Upon successful credential creation, the RP script sends the new credential to the server.
7. The authenticator obtains a biometric or other authorization with some information on the origin that is requesting these keys.

The authenticator in turn returns a response to the Relying Party script. If the user account information provided when creating the credentials, along with some information on the origin that is requesting these keys, 5.

This is flow for when the Relying Party is specifically interested in creating a public key credential with a user-verifying platform authenticator.

1. The user visits example.com and clicks on the login button, which redirects the user to login.example.com.
2. The user enters a username and password to log in. After successful login, the user is redirected back to example.com.
3. The Relying Party script runs the code snippet below.
4. The user agent asks the user whether they are willing to register with the Relying Party using an available platform authenticator.
5. If the user is not willing, terminate this flow.
6. The user is shown appropriate UI and guided in creating a credential using one of the available platform authenticators. Upon successful credential creation, the RP script conveys the new credential to the server.

7. The authenticator obtains a biometric or other authorization with some information on the origin that is requesting these keys.

8. The authenticator returns a response to the client platform, which in turn returns a response to the Relying Party script. If the user account information provided when creating the credentials, along with some information on the origin that is requesting these keys, 8.

12.3. 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 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 account information provided when creating the credentials, along with some information on the origin that is requesting these keys, 7.

The authenticator conveys the new credential to the server.

9. The client platform connects to the authenticator, performing any pairing actions if necessary.
10. The authenticator obtains a biometric or other authorization gesture from the user.
11. The authenticator returns a response to the client platform, which in turn returns a response to the Relying Party script. If the user account information provided when creating the credentials, along with some information on the origin that is requesting these keys, 8.

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 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 account information provided when creating the credentials, along with some information on the origin that is requesting these keys, 7.

The authenticator conveys the new credential to the server.

9. The client platform connects to the authenticator, performing any pairing actions if necessary.
10. The authenticator obtains a biometric or other authorization gesture from the user.
11. The authenticator returns a response to the client platform, which in turn returns a response to the Relying Party script. If the user account information provided when creating the credentials, along with some information on the origin that is requesting these keys, 8.


```javascript
var encoder = new TextEncoder();
var acceptableCredential1 = {
  type: "public-key",
  id: encoder.encode("roses are red, violets are blue
  ");
};
var acceptableCredential2 = {
  type: "public-key",
  id: encoder.encode("!!!!!!!hi there!!!!!!!
  ");
};

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

var options = {
  allowCredentials: [{ type: "public-key" }]
};

navigator.credentials.get({ "publicKey": options }).then(function (assertion) {

  // No acceptable credential or user refused consent. Handle appropriately.
}

// The challenge must be produced by the server, see the Security Considerations

var encoder = new TextEncoder();
var acceptableCredential1 = {
  type: "public-key",
  id: encoder.encode("!!!!!hi there!!!!!
  ");
};
var acceptableCredential2 = {
  type: "public-key",
  id: encoder.encode("roses are red, violets are blue\n  ");
};

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

var options = {
  allowCredentials: [{"type": "public-key"}]
};

navigator.credentials.get({"publicKey": options}).then(function (assertion) {

  // No acceptable credential or user refused consent. Handle appropriately.
}

// The challenge must be produced by the server, see the Security Considerations

var encoder = new TextEncoder();
var acceptableCredential1 = {
  type: "public-key",
  id: encoder.encode("!!!!!!!hi there!!!!!!!
  ");
};
var acceptableCredential2 = {
  type: "public-key",
  id: encoder.encode("roses are red, violets are blue\n  ");
};

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

var options = {
  allowCredentials: [{"type": "public-key"}]
};

navigator.credentials.get({"publicKey": options}).then(function (assertion) {

  // No acceptable credential or user refused consent. Handle appropriately.
}

// The challenge must be produced by the server, see the Security Considerations

var encoder = new TextEncoder();
var acceptableCredential1 = {
  type: "public-key",
  id: encoder.encode("!!!!!hi there!!!!!
  ");
};
var acceptableCredential2 = {
  type: "public-key",
  id: encoder.encode("roses are red, violets are blue\n  ");
};

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

var options = {
  allowCredentials: [{"type": "public-key"}]
};

navigator.credentials.get({"publicKey": options}).then(function (assertion) {

  // No acceptable credential or user refused consent. Handle appropriately.
```

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; that identity is now authenticated.
(If the credential ID is not recognized by the server (e.g., it
has been deregistered due to inactivity) then 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:

```javascript
var encoder = new TextEncoder();
var acceptableCredential1 = {
  type: "public-key",
  id: encoder.encode("roses are red, violets are blue
  ");
};
var acceptableCredential2 = {
  type: "public-key",
  id: encoder.encode("!!!!!!!hi there!!!!!!!
  ");
};

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

var options = {
  allowCredentials: [{ type: "public-key" }]
};

navigator.credentials.get({ "publicKey": options }).then(function (assertion) {

  // 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 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:

```javascript
var encoder = new TextEncoder();
var acceptableCredential1 = {
  type: "public-key",
  id: encoder.encode("!!!!!hi there!!!!!
  ");
};
var acceptableCredential2 = {
  type: "public-key",
  id: encoder.encode("roses are red, violets are blue\n  ");
};

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

var options = {
  allowCredentials: [{"type": "public-key"}]
};

navigator.credentials.get({"publicKey": options}).then(function (assertion) {

  // No acceptable credential or user refused consent. Handle appropriately.
```

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; that identity is now authenticated.
(If the credential ID is not recognized by the server (e.g., it
has been deregistered due to inactivity) then 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:

```javascript
var encoder = new TextEncoder();
var acceptableCredential1 = {
  type: "public-key",
  id: encoder.encode("roses are red, violets are blue
  ");
};
var acceptableCredential2 = {
  type: "public-key",
  id: encoder.encode("!!!!!!!hi there!!!!!!!
  ");
};

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

var options = {
  allowCredentials: [{ type: "public-key" }]
};

navigator.credentials.get({ "publicKey": options }).then(function (assertion) {

  // 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 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:

```javascript
var encoder = new TextEncoder();
var acceptableCredential1 = {
  type: "public-key",
  id: encoder.encode("!!!!!hi there!!!!!
  ");
};
var acceptableCredential2 = {
  type: "public-key",
  id: encoder.encode("roses are red, violets are blue\n  ");
};

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

var options = {
  allowCredentials: [{"type": "public-key"}]
};

navigator.credentials.get({"publicKey": options}).then(function (assertion) {

  // No acceptable credential or user refused consent. Handle appropriately.
```

12.4. Aborting Authentication Operations

The below example shows how a developer may use the AbortSignal parameter to abort a credential registration operation. A similar procedure applies to an authentication operation.

```javascript
const authAbortController = new AbortController();
const authAbortSignal = authAbortController.signal;

authAbortSignal.onabort = function () {
  // Once the page knows the abort started, inform user it is attempting to abort.
}
```

```javascript
var options = {
  // A list of options.
  // A list of options.
  publicKey: options,
  signal: authAbortSignal
}.
```

```javascript
.then(function (attestation) {
  // Register the user.
  authAbortSignal.abort();
})
```

```javascript
.catch(function (error) {
  if (error == "AbortError") {
    // Inform user the credential hasn't been created.
    // Let the server know a key hasn't been created.
  }
});
```

```javascript
// Assume widget shows up whenever auth occurs.
if (widget == "disappear") {
  authAbortSignal.abort();
}
```

12.5. 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.
- Possibility #2 – server deregisters the credential due to inactivity.
- Possibility #3 – user deletes the credential from the device.
- Possibility #4 – the server deregisters the credential during maintenance activity.
- Possibility #5 – in the future, the Relying Party script does not specify this credential in any list of acceptable credentials, and assertions signed by this credential are rejected.
- Possibility #6 – server deletes credential from its database during maintenance activity.
- Possibility #7 – in the future, the Relying Party script does not specify this credential in any list of acceptable credentials, and assertions signed by this credential are rejected.
- Possibility #8 – user employs a device-specific method (e.g., device settings UI) to delete a credential from their device.
- Possibility #9 – from this point on, this credential will not appear in any selection prompts, and no assertions can be generated with it.
- Possibility #10 – sometime later, the server deregisters this credential due to inactivity.

```javascript
const authAbortController = new AbortController();
authAbortSignal.abort();
```

```javascript
navigator.credentials.create({
  publicKey: options,
  signal: authAbortSignal
}.
```

```javascript
.then(function (attestation) {
  // Register the user.
  authAbortSignal.abort();
})
```

```javascript
.catch(function (error) {
  if (error == "AbortError") {
    // Inform user the credential hasn't been created.
    // Let the server know a key hasn't been created.
  }
});
```

```javascript
// Assume widget shows up whenever authentication occurs.
if (widget == "disappear") {
  authAbortSignal.abort();
}
```

12.5. 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.
- Possibility #2 – server deregisters the credential due to inactivity.
- Possibility #3 – user deletes the credential from the device.
- Possibility #4 – the server deregisters the credential during maintenance activity.
- Possibility #5 – in the future, the Relying Party script does not specify this credential in any list of acceptable credentials, and assertions signed by this credential are rejected.
- Possibility #6 – server deletes credential from its database during maintenance activity.
- Possibility #7 – in the future, the Relying Party script does not specify this credential in any list of acceptable credentials, and assertions signed by this credential are rejected.
- Possibility #8 – user employs a device-specific method (e.g., device settings UI) to delete a credential from their device.
- Possibility #9 – from this point on, this credential will not appear in any selection prompts, and no assertions can be generated with it.
- Possibility #10 – sometime later, the server deregisters this credential due to inactivity.

```javascript
const authAbortController = new AbortController();
authAbortSignal.abort();
```

```javascript
navigator.credentials.create({
  publicKey: options,
  signal: authAbortSignal
}.
```

```javascript
.then(function (attestation) {
  // Register the user.
  authAbortSignal.abort();
})
```

```javascript
.catch(function (error) {
  if (error == "AbortError") {
    // Inform user the credential hasn't been created.
    // Let the server know a key hasn't been created.
  }
});
```

```javascript
// Assume widget shows up whenever authentication occurs.
if (widget == "disappear") {
  authAbortSignal.abort();
}
```
13. Security Considerations

13.1. Cryptographic Challenges

As a cryptographic protocol, Web Authentication is dependent upon randomized challenges to avoid replay attacks. Therefore, both challenge’s and challenge’s value, MUST be randomly generated by the Relying Party in an environment they trust (e.g., on the server-side), and the challenge in the client’s response must match what was generated. This should be done in a fashion that does not rely upon a client’s behavior, e.g., the Relying Party should store the challenge temporarily until the operation is complete. Tolerating a mismatch will compromise the security of the protocol.

At this time, this specification does not feature detailed security considerations. However, the [FIDOSecRef] document provides a security analysis which is overall applicable to this specification. Also, the [FIDOAuthnSecReq] document suite defines authenticator security characteristics which are overall applicable for WebAuthn authenticators.

The below subsections comprise the current Web Authentication-specific security considerations.

13.2. Attestation Security Considerations

13.2.1. Attestation Certificate Hierarchy

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

If the attestation root certificate is not dedicated to a single WebAuthn Authenticator device line (i.e., AAGUID), the AAGUID SHOULD be specified in the attestation certificate itself, so that it can be verified against the authenticator data.

13.2.2. Attestation Certificate and Attestation Certificate CA Compromise

When an intermediate CA or a root CA used for issuing attestation certificates is compromised, WebAuthn authenticator attestation keys are still safe although their certificates can no longer be trusted. A WebAuthn Authenticator manufacturer that has recorded the public attestation keys for their devices can issue new attestation certificates for these keys from a new intermediate CA or from a new root CA. If the root CA changes, the 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 this ability, they may need to update the firmware on the devices.
Anonymization CA. Because the Trusted Computing Group (TCG) also
"Privacy CA" is used to refer to what is termed here as an
Note: In various places outside this specification, the term
Note: In various places outside this specification, the term
attestation certificates.

can dynamically generate per-origin attestation keys and
certificate), and combined with a cloud-operated Anonymization CA,
an authenticator can ship with a master attestation key (and
per-origin (similar to the Attestation CA approach). For example,
different attestation keys (and requesting related certificates)
* A WebAuthn authenticator may be capable of dynamically generating
groups. This may serve as guidance about suitable batch sizes.
* A WebAuthn authenticator manufacturer may choose to ship all of
identities of the same user together. This can be mitigated in several
Attestation keys can be used to track users or link various online
14.1. Attestation Privacy
The privacy principles in [FIDO-Privacy-Principles] also apply to this
specification.
14. Privacy Considerations
14.1. Attestation Privacy
Attestation keys can be used to track users or link various online
identities of the same user together. This can 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
if its private key is compromised.
[JAAFProtocol] requires that at least 100,000 devices share the same
attestation certificate in order to produce sufficiently large
groups. This may serve as guidance about suitable batch sizes.
* A WebAuthn authenticator may be capable of dynamically generating
different attestation keys (and requesting related certificates)
per-origin (similar to the Attestation CA approach). For example,
an authenticator can ship with a master attestation key (and
certificate), and combined with a cloud-operated Anonymization CA,
can dynamically generate per-origin attestation keys and
attestation certificates.
* "Privacy CA" is used to refer to what is termed here as an
Anonymization CA. Because the Trusted Computing Group (TCG) also
used the term "Privacy CA" to refer to what the TCG now refers to as an Attestation CA (ACA) [TCG-CMCProfile-AIKCertEnroll], and the envisioned functionality of an Anonymization CA is not firmly established, we are using the term Anonymization CA here to try to mitigate confusion in the specific context of this specification.

* A WebAuthn Authenticator can implement Elliptic Curve based direct anonymous attestation (see [FIDOEcdaaAlgorithm]). Using this scheme, the authenticator generates a blinded attestation signature. This allows the Relying Party to verify the signature using the ECDAA-Issuer public key, but the attestation signature does not serve as a global correlation handle.

### 14.2. Registration Ceremony Privacy

In order to protect users from being identified without consent, implementations of the [[Create]()](origin, options, sameOriginWithAncestors) method need to take care to not leak information that could enable a malicious Relying Party to distinguish between these cases, where "excluded" means that at least one of the credentials listed by the Relying Party in excludeCredentials is bound to the authenticator:

* No authenticators are present.
* At least one authenticator is present, and at least one present authenticator is excluded.

If the above cases are distinguishable, information is leaked by which a malicious Relying Party could identify the user by probing for which credentials are available. For example, one such information leak is if the client returns a failure response as soon as an excluded authenticator becomes available. In this case - especially if the excluded authenticator is a platform authenticator - the Relying Party could detect that the ceremony was canceled before the timeout and before the user could feasibly have canceled it manually, and thus conclude that at least one of the credentials listed in the excludeCredentials parameter is available to the user.

The above is not a concern, however, if the user has consented to create a new credential before a distinguishable error is returned, because in this case the user has confirmed intent to share the information that would be leaked.

### 14.3. Authentication Ceremony Privacy

In order to protect users from being identified without consent, implementations of the [[DiscoverFromExternalSource]()](origin, options, sameOriginWithAncestors) method need to take care to not leak information that could enable a malicious Relying Party to distinguish between these cases, where "named" means that the credential is listed by the Relying Party in allowCredentials:

* A named credential is available, but the user does not consent to use it.

If the above cases are distinguishable, information is leaked by which a malicious Relying Party could identify the user by probing for which credentials are available. For example, one such information leak is if the client returns a failure response as soon as the user denies consent to proceed with an authentication ceremony. In this case the Relying Party could detect that the ceremony was canceled by the user and not the timeout, and thus conclude that at least one of the credentials listed in the allowCredentials parameter is available to the user.

### 15. Acknowledgements

We thank the following people for their reviews of, and contributions to, this specification: Yurii Ackermann, James Barclay, Richard Barnes, Dominic Battr, John Bradley, Domenic Denicola, Rahul Ghosh, Brad Hill, Jing Jin, Wally Jones, Ian Kilpatrick, Axel Nemnker, Yoshikazu Nojima, Kimberly Paulhamus, Adam Powers, Yaron Sheffer, Anne van Kesteren,
We thank Anthony Nadalin, John Fontana, and Richard Barnes for their contributions as co-chairs of the Web Authentication Working Group.

We also thank Wendy Seltzer, Samuel Weiler, and Harry Halpin for their contributions as our W3C Team Contacts.
* Authenticator Response, in 5.2

* Authenticator Selection, in 5.4

* Authenticator Selection Criteria, in 5.4.4

* Authenticator Selection List, in 10.4

* Authenticator session, in 6.2

* Authenticator Transport, in 5.8.4

* Authorization Gesture, in 4

* Base64url Encoding, in 3

* Basic Attestation, in 6.3.3

* Biometric Recognition, in 4

* CBOR, in 3

* Ceremony, in 4

* challenge

+ dfn for PublicKeyCredentialCreationOptions, in 5.4

+ dfn for PublicKeyCredentialRequestOptions, in 5.5

Client, in 4

client data, in 5.8.1

clientDataJSON, in 5.2

clientDataJSONResult

+ dfn for credentialCreationData, in 5.1.3

+ dfn for assertionCreationData, in 5.1.4.1

+ client extension input, in 9.3

+ client extension output, in 9.4

+ client extension processing, in 9.4

+ dfn for credentialCreationData, in 5.1.3

+ dfn for assertionCreationData, in 5.1.4.1

+ [clientExtensions], in 5.9.1

+ [clientExtensionsResults], in 5.1

* Client-Side, in 4

* client-side credential private key storage, in 4

* Client-side-resident Credential Private Key, in 4

* CollectedClientData, in 5.8.1

* [CollectFromCredentialStore][origin, options, sameOriginWithAncestors], in 5.1.4

* Conforming User Agent, in 4

* COSEAlgorithmIdentifier

+ definition of, in 5.8.5

+ (typedef), in 5.3.5

* [[Create][origin, options, sameOriginWithAncestors], in 5.1.3

* Credential ID, in 4

* credentialId, in 6.3.1

* credentialIdLength, in 6.3.1

* credentialIdResult, in 5.1.4.1

* credential key pair, in 4

* credential private key, in 4

* Credential Public Key, in 4

* Credential data claimed to have been used for the attestation, in 6.3.2

* credentialDataResult, in 5.1.4.1

* authenticator extension, in 9

* authenticator extension input, in 9.3

* authenticator extension output, in 9.5

* Authenticator extension processing, in 9.5

* AuthenticatorExtensions, in 5.8.1

* AuthenticatorGetAssertion, in 6.2.3

* AuthenticatorMakeCredential, in 6.2.2

* AuthenticatorTransport, in 5.10.4

* authnSel

+ dict-member for AuthenticationExtensionsClientInputs, in 10.4

+ dict-member for AuthenticationExtensionsClientOutputs, in 10.4

* Authorization Gesture, in 4

* Base64url Encoding, in 3

* Basic, in 6.3.3

* Basic Attestation, in 6.3.3

* Biometric Authenticator, in 4

* Biometric Recognition, in 4

* ble, in 5.10.4

* CBOR, in 3

* Ceremony, in 4

* challenge

+ dict-member for PublicKeyCredentialCreationOptions, in 5.4

+ dict-member for PublicKeyCredentialRequestOptions, in 5.5

+ dict-member for CollectedClientData, in 5.10.1

* Client, in 4

* credential data, in 5.10.1

* credentialDataJSON, in 5.2

* credentialDataJSONResult

+ dfn for credentialCreationData, in 5.1.3

+ dfn for assertionCreationData, in 5.1.4.1

* client extension input, in 9

* client extension output, in 9.4

* client extension processing, in 9.4

* clientExtensionResuts

+ dfn for credentialCreationData, in 5.1.3

+ dfn for assertionCreationData, in 5.1.4.1

* [clientExtensionsResuts], in 5.1

* Client-Side, in 4

* client-side credential private key storage, in 4

* Client-side-resident Credential Private Key, in 4

* CollectedClientData, in 5.10.1

* [CollectFromCredentialStore][origin, options, sameOriginWithAncestors], in 5.1.4

* Conforming User Agent, in 4

* contentType, in 10.3

* COSEAlgorithmIdentifier

+ definition of, in 5.10.5

+ (typedef), in 5.10.5

* [Create][origin, options, sameOriginWithAncestors], in 5.1.3

* Credential ID, in 4

* credentialId, in 6.3.1

* credentialIdLength, in 6.3.1

* credentialIdResult, in 5.1.4.1

* credential key pair, in 4

* credential private key, in 4

* Credential Public Key, in 4

87/126
* credentialPublicKey, in 6.3.1

* "cross-platform", in 5.4.5

* cross-platform, in 5.4.5

* cross-platform attached, in 5.4.5

* DAA, in 6.3.3

* direct, in 5.4.5

* "discouraged", in 5.6

* discouraged, in 5.6

* [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors), in 5.1.4.1

* [[Discovery]], in 5.1

* displayName, in 5.4.3

* ECDAA, in 6.3.3

* ECDAA-Issuer public key, in 8.2

* effective user verification requirement for assertion, in 5.1.4.1

* effective user verification requirement for credential creation, in 5.1.3

* Elliptic Curve based Direct Anonymous Attestation, in 6.3.3

* excludeCredentials, in 5.4

* extension identifier, in 9.1

* extensions

  + dict-member for MakePublicKeyCredentialOptions, in 5.4

  + dict-member for PublicKeyCredentialRequestOptions, in 5.5

* flags, in 6.1

* getClientExtensionResults(), in 5.1

* hashAlgorithm, in 5.6.1

* Hash of the serialized client data, in 5.8.1

* icon, in 5.4.1

* "id"

* + dict-member for PublicKeyCredentialRpEntity, in 5.4.2

* + dict-member for PublicKeyCredentialUserEntity, in 5.4.3

* + dict-member for PublicKeyCredentialDescriptor, in 5.8.3

* [[Identifier]], in 5.1

* indirect, in 5.4.6

* isUserVerifyingPlatformAuthenticatorAvailable(), in 5.1.6

* JSON-serialized client data, in 5.8.1

* MakePublicKeyCredentialOptions, in 5.4

* managing authenticator, in 4

* name, in 5.4.1

* nfc, in 5.8.4

* "origin"

* "origin", in 5.8.1

* platform, in 5.4.5

* "platform", in 5.4.5

* platform attachment, in 5.4.5

* platform authenticators, in 5.4.5

* "preferred", in 5.8.6

* preferred, in 5.8.6

* Privacy CA, in 5.4.3

* credentialPublicKey, in 6.3.1

* credentials map, in 5.6

* "cross-platform", in 5.4.5

* cross-platform, in 5.4.5

* cross-platform attached, in 5.4.5

* DAA, in 6.3.3

* direct, in 5.4.5

* "discouraged", in 5.10.6

* discouraged, in 5.10.6

* [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors), in 5.1.4.1

* [[Discovery]], in 5.1

* DisplayName, in 5.4.3

* ECDAA, in 6.3.3

* ECDAA-Issuer public key, in 8.2

* effective user verification requirement for assertion, in 5.1.4.1

* effective user verification requirement for credential creation, in 5.1.3

* Elliptic Curve based Direct Anonymous Attestation, in 6.3.3

* excludeCredentials, in 5.4

* extension identifier, in 9.1

* extensions

  + dict-member for PublicKeyCredentialCreationOptions, in 5.4

  + dict-member for PublicKeyCredentialRequestOptions, in 5.5

* exts

  + dict-member for AuthenticationExtensionsClientInputs, in 10.5

  + dict-member for AuthenticationExtensionsClientOutputs, in 10.5

* FAR, in 10.9

* flags, in 6.1

* FRH, in 10.9

* getClientExtensionResults(), in 5.1

* Hash of the serialized client data, in 5.10.1

* Human Palatability, in 4

* icon, in 5.4.1

* "id"

* + dfn for public key credential source, in 4

* + dict-member for PublicKeyCredentialReRegistrationOptions, in 5.4.2

* + dict-member for PublicKeyCredentialUserEntity, in 5.4.3

* + dict-member for TokenBinding, in 5.10.1

* + dict-member for PublicKeyCredentialDescriptor, in 5.10.3

* [[Identifier]], in 5.1

* indirect, in 5.4.6

* isUserVerifyingPlatformAuthenticatorAvailable(), in 5.1.6

* JSON-serialized client data, in 5.10.1

* "loc"

* + dict-member for AuthenticationExtensionsClientInputs, in 10.7

* + dict-member for AuthenticationExtensionsClientOutputs, in 10.7

* looking up, in 6.2.1

* managing authenticator, in 4

* name, in 5.4.1

* nfc, in 5.8.4

* "none"

* "none", in 5.4.6

* "origin"

* "origin", in 5.8.1

* platform, in 5.4.5

* "platform", in 5.4.5

* platform attachment, in 5.4.5

* platform authenticators, in 5.4.5

* "preferred", in 5.8.6

* preferred, in 5.8.6

* Privacy CA, in 5.4.3

* credentialPublicKey, in 6.3.1

* credentials map, in 5.6

* "cross-platform", in 5.4.5

* cross-platform, in 5.4.5

* cross-platform attached, in 5.4.5

* DAA, in 6.3.3

* direct, in 5.4.5

* "discouraged", in 5.10.6

* discouraged, in 5.10.6

* [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors), in 5.1.4.1

* [[Discovery]], in 5.1

* DisplayName, in 5.4.3

* ECDAA, in 6.3.3

* ECDAA-Issuer public key, in 8.2

* effective user verification requirement for assertion, in 5.1.4.1

* effective user verification requirement for credential creation, in 5.1.3

* Elliptic Curve based Direct Anonymous Attestation, in 6.3.3

* excludeCredentials, in 5.4

* extension identifier, in 9.1

* extensions

  + dict-member for PublicKeyCredentialCreationOptions, in 5.4

  + dict-member for PublicKeyCredentialRequestOptions, in 5.5

* exts

  + dict-member for AuthenticationExtensionsClientInputs, in 10.5

  + dict-member for AuthenticationExtensionsClientOutputs, in 10.5

* FAR, in 10.9

* flags, in 6.1

* FRH, in 10.9

* getClientExtensionResults(), in 5.1

* Hash of the serialized client data, in 5.10.1

* Human Palatability, in 4

* icon, in 5.4.1

* "id"

* + dfn for public key credential source, in 4

* + dict-member for PublicKeyCredentialReRegistrationOptions, in 5.4.2

* + dict-member for PublicKeyCredentialUserEntity, in 5.4.3

* + dict-member for TokenBinding, in 5.10.1

* + dict-member for PublicKeyCredentialDescriptor, in 5.10.3

* [[Identifier]], in 5.1

* indirect, in 5.4.6

* isUserVerifyingPlatformAuthenticatorAvailable(), in 5.1.6

* JSON-serialized client data, in 5.10.1

* "loc"

* + dict-member for AuthenticationExtensionsClientInputs, in 10.7

* + dict-member for AuthenticationExtensionsClientOutputs, in 10.7

* looking up, in 6.2.1

* managing authenticator, in 4

* name, in 5.4.1

* nfc, in 5.8.4

* "none"

* "none", in 5.4.6

* "origin"

* "origin", in 5.8.1

* platform, in 5.4.5

* "platform", in 5.4.5

* platform attachment, in 5.4.5

* platform authenticators, in 5.4.5

* "preferred", in 5.8.6

* preferred, in 5.8.6

* Privacy CA, in 5.4.3
521:  * publicKeyCredParams, in 5.4
522:  * publicKey
523:  + dict-member for CredentialCreationOptions, in 5.1.1
524:  + dict-member for CredentialRequestOptions, in 5.1.2
525:  + key, in 5.8.2
526:  + PublicKeyCredential, in 4
527:  + PublicKeyCredentialDescriptor, in 5.8.3
528:  + TokenBinding
529:  + tokenBindingId, in 5.8.1
5210:  + transports, in 5.8.3
5211:  + txAuthGeneric
5212:  + txAuthGenericArg, in 10.3
5213:  + dict-member for AuthenticationExtensionsClientOutputs, in 10.2
5214:  + dict-member for AuthenticationExtensionsClientInputs, in 10.3
5215:  + txAuthGenericArg, in 10.3
5216:  + txAuthSimple
5217:  + dict-member for AuthenticationExtensionsClientOutputs, in 10.2
+ [collectFromCredentialStore] (origin, options, [signal (for CredentialCreationOptions)])
+ [store] ()
+ Credential

Terms defined by reference

+ [CREDENTIAL-MANAGEMENT-1] defines the following terms:
+ Credential
+ CredentialCreationOptions
+ CredentialRequestOptions
+ [CredentialContainer]

+ [CollectFromCredentialStore] (origin, options, [signal (for CredentialCreationOptions)])
+ [Store] (credential, sameOriginWithAncestors)

+ [type]
+ create()
+ credential
+ [credentialSource]
+ get()
+ id
+ [same-origin with its ancestors]
+ [signal (for CredentialCreationOptions)]
+ [signal (for CredentialRequestOptions)]
+ [store] ()

+ [type]
+ [user mediation]

+ [DOM] defines the following terms:
+ AbortController
+ [AbortedFlag]
+ [document]

+ [ECMAScript] defines the following terms:
+ [arrayBuffer] %
+ [internal method]
+ [internal slot]
+ [stringify]
[FIDO-U2F-Message-Formats]

[FIDO-EcdaaAlgorithm]

[FIDO-Reg]
R. Lindemann; R. Baghdasaryan; B. Hill. FIDO UAF Registry of Predefined Values. FIDO Alliance Proposed Standard. URL: https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-reg-v1.0-ps-20141208.html

[HTML]
Anne van Kesteren; et al. HTML Standard. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[HTML52]
Steve Faulkner; et al. HTML 5.2. 2 November 2017. PR. URL: https://www.w3.org/TR/html52/

[IANA-COSE-ALGS-REG]
IANA CBOR Object Signing and Encryption (COSE) Algorithms Registry. URL: https://www.iana.org/assignments/cose/cose.xhtml#algorithms

[INFRA]
Anne van Kesteren; Domenic Denicola. Infra Standard. Living Standard. URL: https://infra.spec.whatwg.org/

[MIXED-CONTENT]
Mike West. Mixed Content. 2 August 2016. CR. URL: https://www.w3.org/TR/mixed-content/

[PAGE-VISIBILITY]

[RFC2119]

[RFC4648]

[RFC5234]

[RFC5890]

[RFC7049]
interface AuthenticatorAssertionResponse : AuthenticatorResponse {
  [SecureContext, Exposed=Window]
  [SameObject] readonly attribute ArrayBuffer      attestationObject;
  [SameObject] readonly attribute ArrayBuffer      clientDataJSON;
}

interface AuthenticatorResponse {
  [SecureContext, Exposed=Window]
  [SecureContext, Exposed=Window]
  static Promise < boolean > isUserVerifyingPlatformAuthenticatorAvailable();
}

partial dictionary CredentialRequestOptions {
  publicKey;
}

partial dictionary CredentialCreationOptions {
  publicKey;
}

partial interface PublicKeyCredential {
  static Promise < boolean > isUserVerifyingPlatformAuthenticatorAvailable();
}

IDL Index

SecureContext, Exposed=Window
interface PublicKeyCredential : Credential {
  [SameObject] readonly attribute ArrayBuffer      rawId;
  AuthenticationExtensions getClientExtensionResults();
}

partial dictionary CredentialCreationOptions {
  publicKey;
};

partial dictionary CredentialRequestOptions {
  publicKey;
};

interface PublicKeyCredentialRequestOptions {
  [SameObject] readonly attribute ArrayBuffer      rawId;
  [SameObject] readonly attribute AuthenticatorResponse response;
  AuthenticationExtensionsClientOutputs getClientExtensionResults();
};

interface PublicKeyCredential : Credential {
  static Promise < boolean > isUserVerifyingPlatformAuthenticatorAvailable();
}

IDL Index

SecureContext, Exposed=Window
interface PublicKeyCredential : Credential {
  [SameObject] readonly attribute ArrayBuffer      rawId;
  [SameObject] readonly attribute AuthenticatorResponse response;
  AuthenticationExtensionsClientOutputs getClientExtensionResults();
};

partial dictionary CredentialCreationOptions {
  publicKey;
};

partial dictionary CredentialRequestOptions {
  publicKey;
};

partial interface PublicKeyCredential {
  static Promise < boolean > isUserVerifyingPlatformAuthenticatorAvailable();
}

IDL Index

SecureContext, Exposed=Window
interface PublicKeyCredential : Credential {
  [SameObject] readonly attribute ArrayBuffer      rawId;
  [SameObject] readonly attribute AuthenticatorResponse response;
  AuthenticationExtensionsClientOutputs getClientExtensionResults();
};

interface PublicKeyCredential : Credential {
  static Promise < boolean > isUserVerifyingPlatformAuthenticatorAvailable();
}

IDL Index

SecureContext, Exposed=Window
interface PublicKeyCredential : Credential {
  [SameObject] readonly attribute ArrayBuffer      rawId;
  [SameObject] readonly attribute AuthenticatorResponse response;
  AuthenticationExtensionsClientOutputs getClientExtensionResults();
};

interface PublicKeyCredential : Credential {
  static Promise < boolean > isUserVerifyingPlatformAuthenticatorAvailable();
}

IDL Index

SecureContext, Exposed=Window
interface PublicKeyCredential : Credential {
  [SameObject] readonly attribute ArrayBuffer      rawId;
  [SameObject] readonly attribute AuthenticatorResponse response;
  AuthenticationExtensionsClientOutputs getClientExtensionResults();
};

interface PublicKeyCredential : Credential {
  static Promise < boolean > isUserVerifyingPlatformAuthenticatorAvailable();
}

IDL Index

SecureContext, Exposed=Window
interface PublicKeyCredential : Credential {
  [SameObject] readonly attribute ArrayBuffer      rawId;
  [SameObject] readonly attribute AuthenticatorResponse response;
  AuthenticationExtensionsClientOutputs getClientExtensionResults();
};

interface PublicKeyCredential : Credential {
  static Promise < boolean > isUserVerifyingPlatformAuthenticatorAvailable();
}

IDL Index

SecureContext, Exposed=Window
interface PublicKeyCredential : Credential {
  [SameObject] readonly attribute ArrayBuffer      rawId;
  [SameObject] readonly attribute AuthenticatorResponse response;
  AuthenticationExtensionsClientOutputs getClientExtensionResults();
};

interface PublicKeyCredential : Credential {
  static Promise < boolean > isUserVerifyingPlatformAuthenticatorAvailable();
}

IDL Index

SecureContext, Exposed=Window
interface PublicKeyCredential : Credential {
  [SameObject] readonly attribute ArrayBuffer      rawId;
  [SameObject] readonly attribute AuthenticatorResponse response;
  AuthenticationExtensionsClientOutputs getClientExtensionResults();
};

interface PublicKeyCredential : Credential {
  static Promise < boolean > isUserVerifyingPlatformAuthenticatorAvailable();
}

IDL Index

SecureContext, Exposed=Window
interface PublicKeyCredential : Credential {
  [SameObject] readonly attribute ArrayBuffer      rawId;
  [SameObject] readonly attribute AuthenticatorResponse response;
  AuthenticationExtensionsClientOutputs getClientExtensionResults();
};

interface PublicKeyCredential : Credential {
  static Promise < boolean > isUserVerifyingPlatformAuthenticatorAvailable();
}

IDL Index

SecureContext, Exposed=Window
interface PublicKeyCredential : Credential {
  [SameObject] readonly attribute ArrayBuffer      rawId;
  [SameObject] readonly attribute AuthenticatorResponse response;
  AuthenticationExtensionsClientOutputs getClientExtensionResults();
};

interface PublicKeyCredential : Credential {
  static Promise < boolean > isUserVerifyingPlatformAuthenticatorAvailable();
}
UserVerificationRequirement: userVerification = "preferred";

sequence<PublicKeyCredentialDescriptor>: allowCredentials = [];

USVString: rpId;

unsigned long: timeout;

required BufferSource: challenge;

dictionary PublicKeyCredentialRequestOptions {
  required COSEAlgorithmIdentifier alg;
  required COSEAlgorithmIdentifier alg;
};

dictionary MakePublicKeyCredentialOptions {
  required PublicKeyCredentialRpEntity rp;
  required PublicKeyCredentialUserEntity user;
  required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
};

dictionary PublicKeyCredentialEntity {
  required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
};

dictionary PublicKeyCredentialCreationOptions {
  required PublicKeyCredentialType type;
  required COSEAlgorithmIdentifier alg;
  required PublicKeyCredentialUserEntity user;
  required PublicKeyCredentialRpEntity rp;
  required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
};

dictionary PublicKeyCredentialRpEntity: PublicKeyCredentialEntity {
  required DOMString id;
  required DOMString displayName;
};

dictionary PublicKeyCredentialUserEntity: PublicKeyCredentialEntity {
  required BufferSource challenge;
  required DOMString name;
  required DOMString displayname;
};

dictionary AuthenticatorSelectionCriteria {
  required BufferSource challenge;
  required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
};

dictionary AuthenticatorAttachment {
  "platform" // Platform attachment
  "cross-platform" // Cross-platform attachment
};

dictionary AuthenticatorSelectionConveyancePreference {
  "none",
  "indirect",
  "direct"
};

dictionary PublicKeyCredentialRequestOptions {
  required BufferSource challenge;
  required COSEAlgorithmIdentifier alg;
};

dictionary PublicKeyCredentialCreationOptions {
  required PublicKeyCredentialType type;
  required COSEAlgorithmIdentifier alg;
  required PublicKeyCredentialUserEntity user;
  required PublicKeyCredentialRpEntity rp;
  required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
};

typedef record<DOMString, DOMString> AuthenticationExtensionsAuthenticatorInputs;

dictionary AuthenticationExtensionsClientOutputs {
  AuthenticationExtensionsClientInputs extensions;
};

dictionary AuthenticationExtensionsClientInputs {
  AuthenticationExtensionsClientInputs extensions;
};

dictionary PublicKeyCredentialEntity {
  required DOMString name;
  required DOMString icon;
};

dictionary PublicKeyCredentialRpEntity: PublicKeyCredentialEntity {
  required DOMString id;
  required DOMString displayName;
};

dictionary PublicKeyCredentialUserEntity: PublicKeyCredentialEntity {
  required BufferSource challenge;
  required DOMString name;
  required DOMString displayname;
};

dictionary AuthenticatorSelectionCriteria {
  required BufferSource challenge;
  required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
};

dictionary AuthenticatorAttachment {
  "platform" // Platform attachment
  "cross-platform" // Cross-platform attachment
};

dictionary AuthenticatorSelectionConveyancePreference {
  "none",
  "indirect",
  "direct"
};

dictionary PublicKeyCredentialRequestOptions {
  required BufferSource challenge;
  required COSEAlgorithmIdentifier alg;
  required sequence<PublicKeyCredentialParameters> allowCredentials = [];
};

dictionary PublicKeyCredentialCreationOptions {
  required PublicKeyCredentialType type;
  required COSEAlgorithmIdentifier alg;
  required PublicKeyCredentialUserEntity user;
  required PublicKeyCredentialRpEntity rp;
  required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
};

typedef record<DOMString, DOMString> AuthenticationExtensionsAuthenticatorInputs;

dictionary AuthenticationExtensionsClientOutputs {
  AuthenticationExtensionsClientInputs extensions;
};

dictionary PublicKeyCredentialEntity {
  required DOMString name;
  required DOMString icon;
};

dictionary PublicKeyCredentialRpEntity: PublicKeyCredentialEntity {
  required DOMString id;
  required DOMString displayName;
};

dictionary PublicKeyCredentialUserEntity: PublicKeyCredentialEntity {
  required BufferSource challenge;
  required DOMString name;
  required DOMString displayname;
};

dictionary AuthenticatorSelectionCriteria {
  required BufferSource challenge;
  required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
};

dictionary AuthenticatorAttachment {
  "platform" // Platform attachment
  "cross-platform" // Cross-platform attachment
};

dictionary AuthenticatorSelectionConveyancePreference {
  "none",
  "indirect",
  "direct"
};

dictionary PublicKeyCredentialRequestOptions {
  required BufferSource challenge;
  required COSEAlgorithmIdentifier alg;
  required sequence<PublicKeyCredentialParameters> allowCredentials = [];
};

dictionary PublicKeyCredentialCreationOptions {
  required PublicKeyCredentialType type;
  required COSEAlgorithmIdentifier alg;
  required PublicKeyCredentialUserEntity user;
  required PublicKeyCredentialRpEntity rp;
  required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
};

dictionary MakePublicKeyCredentialOptions {
  required PublicKeyCredentialRpEntity rp;
  required PublicKeyCredentialUserEntity user;
  required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
};

dictionary PublicKeyCredentialEntity {
  required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
};

dictionary PublicKeyCredentialCreationOptions {
  required PublicKeyCredentialType type;
  required COSEAlgorithmIdentifier alg;
  required PublicKeyCredentialUserEntity user;
  required PublicKeyCredentialRpEntity rp;
  required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
};

dictionary AuthenticationExtensionsAuthenticatorInputs {
  AuthenticationExtensionsAuthenticatorInputs extensions;
};

dictionary AuthenticationExtensionsClientOutputs {
  AuthenticationExtensionsClientInputs extensions;
};

dictionary PublicKeyCredentialEntity {
  required DOMString name;
  required DOMString icon;
};

dictionary PublicKeyCredentialRpEntity: PublicKeyCredentialEntity {
  required DOMString id;
  required DOMString displayName;
};

dictionary PublicKeyCredentialUserEntity: PublicKeyCredentialEntity {
  required BufferSource challenge;
  required DOMString name;
  required DOMString displayname;
};

dictionary AuthenticatorSelectionCriteria {
  required BufferSource challenge;
  required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
};

dictionary AuthenticatorAttachment {
  "platform" // Platform attachment
  "cross-platform" // Cross-platform attachment
};

dictionary AuthenticatorSelectionConveyancePreference {
  "none",
  "indirect",
  "direct"
};

dictionary PublicKeyCredentialRequestOptions {
  required BufferSource challenge;
  required COSEAlgorithmIdentifier alg;
  required sequence<PublicKeyCredentialParameters> allowCredentials = [];
};

dictionary PublicKeyCredentialCreationOptions {
  required PublicKeyCredentialType type;
  required COSEAlgorithmIdentifier alg;
  required PublicKeyCredentialUserEntity user;
  required PublicKeyCredentialRpEntity rp;
  required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
};

typedef record<DOMString, DOMString> AuthenticationExtensionsAuthenticatorInputs;

dictionary AuthenticationExtensionsClientOutputs {
  AuthenticationExtensionsClientInputs extensions;
};

dictionary PublicKeyCredentialEntity {
  required DOMString name;
  required DOMString icon;
};

dictionary PublicKeyCredentialRpEntity: PublicKeyCredentialEntity {
  required DOMString id;
  required DOMString displayName;
};

dictionary PublicKeyCredentialUserEntity: PublicKeyCredentialEntity {
  required BufferSource challenge;
  required DOMString name;
  required DOMString displayname;
};

dictionary AuthenticatorSelectionCriteria {
  required BufferSource challenge;
  required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
};

dictionary AuthenticatorAttachment {
  "platform" // Platform attachment
  "cross-platform" // Cross-platform attachment
};

dictionary AuthenticatorSelectionConveyancePreference {
  "none",
  "indirect",
  "direct"
};

dictionary PublicKeyCredentialRequestOptions {
  required BufferSource challenge;
  required COSEAlgorithmIdentifier alg;
  required sequence<PublicKeyCredentialParameters> allowCredentials = [];
};

dictionary PublicKeyCredentialCreationOptions {
  required PublicKeyCredentialType type;
  required COSEAlgorithmIdentifier alg;
  required PublicKeyCredentialUserEntity user;
  required PublicKeyCredentialRpEntity rp;
  required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
};

typedef record<DOMString, DOMString> AuthenticationExtensionsAuthenticatorInputs;

dictionary AuthenticationExtensionsClientOutputs {
  AuthenticationExtensionsClientInputs extensions;
};

dictionary PublicKeyCredentialEntity {
  required DOMString name;
  required DOMString icon;
};

dictionary PublicKeyCredentialRpEntity: PublicKeyCredentialEntity {
  required DOMString id;
  required DOMString displayName;
};

dictionary PublicKeyCredentialUserEntity: PublicKeyCredentialEntity {
  required BufferSource challenge;
  required DOMString name;
  required DOMString displayname;
};

dictionary AuthenticatorSelectionCriteria {
  required BufferSource challenge;
  required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
};

dictionary AuthenticatorAttachment {
  "platform" // Platform attachment
  "cross-platform" // Cross-platform attachment
};

dictionary AuthenticatorSelectionConveyancePreference {
  "none",
  "indirect",
  "direct"
};

dictionary PublicKeyCredentialRequestOptions {
  required BufferSource challenge;
  required COSEAlgorithmIdentifier alg;
  required sequence<PublicKeyCredentialParameters> allowCredentials = [];
};

dictionary PublicKeyCredentialCreationOptions {
  required PublicKeyCredentialType type;
  required COSEAlgorithmIdentifier alg;
  required PublicKeyCredentialUserEntity user;
  required PublicKeyCredentialRpEntity rp;
  required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
};
enum UserVerificationRequirement { "required", "preferred", "discouraged" };

dictionary PublicKeyCredentialDescriptor {
    required PublicKeyCredentialType type;
    required BufferSource id;
    sequence<AuthenticatorTransport> transports;
    required DOMString hashAlgorithm;
    AuthenticationExtensions clientExtensions;
    AuthenticationExtensions authenticatorExtensions;
};

dictionary CollectedClientData {
    required DOMString challenge;
    required DOMString origin;
    DOMString tokenBindingId;
    AuthenticationExtensions tokenBinding;
    AuthenticationExtensions authenticatorExtensions;
};

enum PublicKeyCredentialType {
    "public-key"
};

dictionary txAuthGenericArg {
    USVString txAuthSimple;
    "ble";
    "nfc";
};

partial dictionary AuthenticationExtensionsClientInputs {
    USVString appid;
    boolean appid;
};

partial dictionary AuthenticationExtensionsClientOutputs {
    USVString txAuthSimple;
    USVString txAuthGenericArg;
};
The definitions of "lifetime of" and "becomes available" are intended
to represent how devices are hotplugged into (USB) or discovered by
the client SHOULD terminate this algorithm with a
violates the privacy requirements of the attestation type it is using,
@balfanz wishes to add to the "direct" case: If the authenticator
need to define "blinding". See also #462:
<https://github.com/w3c/webauthn/issues/694> RET
@balfanz wishes to add to the "direct" case: If the authenticator
violates the privacy requirements of the attestation type it is using,
the client SHOULD terminate this algorithm with a
"AttestationNotPrivateError", RET
The definitions of "lifetime of" and "becomes available" are intended
to represent how devices are hotplugged into (USB) or discovered by

typedef sequence<AAGUID> AuthenticatorSelectionList;

typedef BufferSource AAGUID;

partial dictionary AuthenticationExtensionsClientInputs {
  AuthenticatorSelectionList authnSel;
};

typedef sequence<unsigned long> UvmEntry;

dictionary authenticatorBiometricPerfBounds{
  float FAR;
  float FRR;
};

dictionary authenticatorBiometricPerfBounds{
  float FAR;
  float FRR;
};
5800 (NFC) browsers, and are underspecified. Resolving this with good
5801 definitions or some other means will be addressed by resolving issue
5802 #613. RET
5803 The foregoing step may be incorrect, in that we are attempting to
5804 create savedCredentialId here and use it later below, and we do not
5805 have a method in which to allocate a place for it. Perhaps this is good
5806 enough? addendum: @cjones feels the above step is likely good enough.
5807 RET
5808 The WHATWG HTML WG is discussing whether to provide a hook when a
5809 context gains or loses focuses. If a hook is provided, the
5810 above paragraph will be updated to include the hook. See WHATWG HTML WG
5811 Issue #2711 for more details. RET
5812
5813 #base64url-encodingReferenced in:
5814 * 5.1. PublicKeyCredential Interface
5815 * 5.1.3. Create a new credential - PublicKeyCredential's
5816 [[Create]](origin, options, sameOriginWithAncestors) method (2)
5817 * 5.4.6. Attestation Conveyance Preference enumeration (enum
5818 AttestationConveyancePreference) (2)
5819 * 9.2. Defining extensions (2)
5820 * 9.3. Extending request parameters (2)
5821 * 9.5. Authenticator extension processing (2)
5822
5823 #cborReferenced in:
5824 * 5.1.3. Create a new credential - PublicKeyCredential's
5825 [[Create]](origin, options, sameOriginWithAncestors) method
5826 * 5.4.6. Attestation Conveyance Preference enumeration (enum
5827 AttestationConveyancePreference) (2)
5828 * 9.2. Defining extensions (2)
5829 * 9.3. Extending request parameters (2)
5830 * 9.5. Authenticator extension processing (2)
5831
5832 #attestationReferenced in:
5833 * 4. Terminology (2)
5834 * 5.4.6. Attestation Conveyance Preference enumeration (enum
5835 AttestationConveyancePreference) (2)
5836 * 6. WebAuthn Authenticator model (2)
5837 * 6.3. Attestation (2) (3) (4)
5838 * 11.1. WebAuthn Attestation Statement Format (dictionary
5839 CollectedClientData)
5840
5841 #attestation-certificateReferenced in:
5842 * 5.1.3. Create a new credential - PublicKeyCredential's
5843 [[Create]](origin, options, sameOriginWithAncestors) method
5844 * 6.3.1. TPM attestation statement certificate requirements
5845
5846 #attestation-key-pairReferenced in:
5847 * 4. Terminology (2)
5848 * 6.3. Attestation
5849
5850 #attestation-private-keyReferenced in:
5851 * 4. Terminology (2)
5852 * 6.3. Attestation
5853 * 6.3.3. Attestation Types
5854
5855 #assertionReferenced in:
5856 * 7.1. Registering a new credential
5857 * 7.2. Verifying an authentication assertion (2)
5858
5859 #613. RET
5860 The foregoing step may be incorrect, in that we are attempting to
5861 create savedCredentialId here and use it later below, and we do not
5862 have a method in which to allocate a place for it. Perhaps this is good
5863 enough? addendum: @cjones feels the above step is likely good enough.
5864 RET
5865 The WHATWG HTML WG is discussing whether to provide a hook when a
5866 context gains or loses focuses. If a hook is provided, the
5867 above paragraph will be updated to include the hook. See WHATWG HTML WG
5868 Issue #2711 for more details. RET
5869
5870 #base64url-encodingReferenced in:
5871 * 5.1. PublicKeyCredential Interface
5872 * 5.1.3. Create a new credential - PublicKeyCredential's
5873 [[Create]](origin, options, sameOriginWithAncestors) method (2)
5874 * 5.4.6. Attestation Conveyance Preference enumeration (enum
5875 AttestationConveyancePreference) (2)
5876 * 9.2. Defining extensions (2)
5877 * 9.3. Extending request parameters (2)
5878 * 9.5. Authenticator extension processing (2)
5879
5880 #cborReferenced in:
5881 * 5.1.3. Create a new credential - PublicKeyCredential's
5882 [[Create]](origin, options, sameOriginWithAncestors) method
5883 * 5.4.6. Attestation Conveyance Preference enumeration (enum
5884 AttestationConveyancePreference) (2)
5885 * 9.2. Defining extensions (2)
5886 * 9.3. Extending request parameters (2)
5887 * 9.5. Authenticator extension processing (2)
5888
5889 #attestationReferenced in:
5890 * 4. Terminology (2)
5891 * 5.4.6. Attestation Conveyance Preference enumeration (enum
5892 AttestationConveyancePreference) (2)
5893 * 6. WebAuthn Authenticator model (2)
5894 * 6.3. Attestation (2) (3) (4)
5895 * 11.1. WebAuthn Attestation Statement Format (dictionary
5896 CollectedClientData)
5897
5898 #attestation-certificateReferenced in:
5899 * 5.1.3. Create a new credential - PublicKeyCredential's
6000 [[Create]](origin, options, sameOriginWithAncestors) method
6001 * 6.3.1. TPM attestation statement certificate requirements
6002
6003 #attestation-key-pairReferenced in:
6004 * 4. Terminology (2)
6005 * 6.3. Attestation
6006 * 6.3.3. Attestation Types
6007
6008 #attestation-private-keyReferenced in:
8.4. Android Key Attestation Statement Format

7.1. Registering a new credential

6.3. Attestation

6.2. Packed Attestation Statement Format

6. WebAuthn Authenticator Model

5.8. Attestation

5.7. Authenticator Attachment enumeration (enum AuthenticatorAttachment)

5.6. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)

5.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)

5.4.1. Public Key Entity Description (dictionary PublicKeyCredentialEntity)

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

5.4.5. Authenticator Attachment enumeration (enum AuthenticatorAttachment)

5.4.6. Attestation Conveyance Preference enumeration (enum AttestationConveyancePreference)

5.4.5. Authenticator Attachment enumeration (enum AuthenticatorAttachment)

5.3. Attestation (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13)

5.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)

5.2.1. Information about Public Key Credential (interface PublicKeyCredential)

5.1.4.1. PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)

5.1.4. Public Key Credential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method (2) (3) (4) (5)

5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)

5.1.2. Information about Public Key Credential (interface PublicKeyCredential)

5.1.1. Use Cases

5.1.0. Authenticator Attachment (enum AuthenticatorAttachment)

5.0.2. Web Authentication API (2) (3) (4)

5.0.1. PublicKeyCredential Interface

5.0.0.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)

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

5.0.0.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)

5.0.0.2. Backwards Compatibility with FIDO U2F

5.0.0.1. Public Key Entity Description (dictionary PublicKeyCredentialEntity)

5.0.0. Authenticator Responses (interface AuthenticatorResponse)

5.0. Authenticator model (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13) (14)


3. Security Considerations

2. WebAuthn Authenticator model (2) (3) (4) (5) (6)

1. WebAuthn Authenticator model

1.1. Use Cases

1. Introduction (2)

1.1. Use Cases

1.0. Authenticator Attachment (enum AuthenticatorAttachment)

1.0.0. Authenticator Attachment (enum AuthenticatorAttachment)


8.4. Android Key Attestation Statement Format

8.2. Packed Attestation Statement Format

8.1. Introduction (2) (3) (4)

7.2. Verifying an authentication assertion (2) (3) (4)

7.1. Registering a new credential

7.0.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)

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

7.0.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)

7.0.2. Authenticator Attachment enumeration (enum AuthenticatorAttachment)

7.0.1. Public Key Entity Description (dictionary PublicKeyCredentialEntity)

7.0. Authenticator Responses (interface AuthenticatorResponse)


6.3. Attestation (2) (3) (4) (5) (6) (7) (8) (9)

6.3.2. Attestation Statement Formats

6.2.2. The authenticatorMakeCredential operation (2) (3) (4)

6.2.1. Lookup Credential Source by Credential ID algorithm

6.1. Authenticator data

6.0.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)

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

6.0.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)

6.0.2. Backwards Compatibility with FIDO U2F

6.0.1. Public Key Entity Description (dictionary PublicKeyCredentialEntity)

6.0. Authenticator Responses (interface AuthenticatorResponse)


5.4.5. Authenticator Attachment enumeration (enum AuthenticatorAttachment)

5.4.4. Attestation Conveyance Preference enumeration (enum AttestationConveyancePreference)

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

5.4.2. The authenticatorGetAssertion operation (2) (3) (4)

5.4.1. Public Key Entity Description (dictionary PublicKeyCredentialEntity)

5.4.0. Authenticator Responses (interface AuthenticatorResponse)

5.3. Attestation (2) (3) (4) (5) (6) (7) (8) (9)

5.3.2. Attestation Statement Formats

5.3.1. Privacy

5.3.0.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)

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

5.3.0.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)

5.3.0.2. Backwards Compatibility with FIDO U2F

5.3.0.1. Public Key Entity Description (dictionary PublicKeyCredentialEntity)

5.3.0. Authenticator Responses (interface AuthenticatorResponse)

5.3. Authenticator model (2) (3) (4) (5) (6) (7) (8)

5.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)

5.2.1. Information about Public Key Credential (interface PublicKeyCredential)

5.2. Authenticator Responses (interface AuthenticatorResponse)

5.1.4.1. PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2) (3) (4) (5)

5.1.4. Public Key Credential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method (2) (3) (4) (5)

5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)

5.1.2. Information about Public Key Credential (interface PublicKeyCredential)

5.1.1. Use Cases

5.1.0. Authenticator Attachment (enum AuthenticatorAttachment)

5.0.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)

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

5.0.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)

5.0.2. Authenticator Attachment enumeration (enum AuthenticatorAttachment)

5.0.1. Public Key Entity Description (dictionary PublicKeyCredentialEntity)

5.0.0.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)

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

5.0.0.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)

5.0.0.2. Backwards Compatibility with FIDO U2F

5.0.0.1. Public Key Entity Description (dictionary PublicKeyCredentialEntity)

5.0.0. Authenticator Responses (interface AuthenticatorResponse)


3. Security Considerations

2. WebAuthn Authenticator model (2) (3) (4) (5) (6)

1. WebAuthn Authenticator model

1.1. Use Cases

1. Introduction (2)

1.1. Use Cases

1.0. Authenticator Attachment (enum AuthenticatorAttachment)

1.0.0. Authenticator Attachment (enum AuthenticatorAttachment)


8.4. Android Key Attestation Statement Format

8.2. Packed Attestation Statement Format

8.1. Introduction (2) (3) (4)

7.2. Verifying an authentication assertion (2) (3) (4)

7.1. Registering a new credential
5.1.3. Create a new credential - PublicKeyCredential's

5.2.1. Information about Public Key Credential (interface

AuthenticatorAttestationResponse)

5.10.3. Credential Descriptor (dictionary

PublicKeyCredentialDescriptor)

6.2.1. Lookup Credential Source by Credential ID algorithm

6.2.2. The authenticatorMakeCredential operation

6.2.3. The authenticatorGetAssertion operation

6.3.1. Attested credential data

6.7.1. Registering a new credential

8.6. FIDO U2F Attestation Statement Format

12.1. Registration

12.3. Authentication (2) (3)

13.3. credentialId Unsigned (2) (3)

13.3. credentialKeyId Unsigned (2) (3)

4. Terminology (2) (3)

5.1. PublicKeyCredential Interface

5.2.1. Information about Public Key Credential (interface

AuthenticatorAttestationResponse)

5.2.2. Web Authentication Assertion (interface

PublicKeyCredentialEntity) (2)

5.4.1. Public Key Entity Description (dictionary

PublicCredentialEntity) (2)

5.10.3. Credential Descriptor (dictionary

PublicKeyCredentialDescriptor)

6.2.1. Lookup Credential Source by Credential ID algorithm

6.2.2. The authenticatorMakeCredential operation

6.2.3. The authenticatorGetAssertion operation

6.3.1. Attested credential data

6.7.1. Registering a new credential

8.6. FIDO U2F Attestation Statement Format

12.1. Registration

12.3. Authentication (2) (3)

13.3. credentialId Unsigned (2) (3)

#credential-public-keyReferenced in:

#public-key-credential-source-userhandleReferenced in:

#public-key-credential-source-rpidReferenced in:

#public-key-credential-source-privatekeyReferenced in:

#public-key-credential-source-idReferenced in:

#public-key-credential-source-typeReferenced in:

[Create](origin, options, sameOriginWithAncestors) method

5.1.3. Create a new credential - PublicKeyCredential's

[[Create]](origin, options, sameOriginWithAncestors) method
5.1.3. Create a new credential - PublicKeyCredential's
sameOriginWithAncestors) method (2) (3)

* 5. Web Authentication API (2) (3) (4)

4. Terminology (2) (3) (4) (5) (6) (7) (8)

1. Introduction (2) (3) (4) (5)

* 5.1. PublicKeyCredential Interface (2) (3) (4)

5.1.3. Create a new credential - PublicKeyCredential's
sameOriginWithAncestors) method (2) (3) (4)

* 5.1.4.1. PublicKeyCredential's
sameOriginWithAncestors) method (2)

5.1.4. User about Public Credential (interface
AuthenticatorAttestationResponse)

PublicCredentialEntity (dictionary
AuthenticatorSelectionCriteria)

* 5.4.5. Authenticator Selection Criteria (dictionary
AuthenticatorSelectionCriteria)

4. Terminology (2) (3) (4) (5) (6) (7) (8)

1. Introduction (2) (3) (4) (5)

* 5.1.4.1. PublicKeyCredential's
sameOriginWithAncestors) method (2)

5.1.4. User about Public Credential (interface
AuthenticatorAttestationResponse)

PublicCredentialEntity (dictionary
AuthenticatorSelectionCriteria)

* 5.4.5. Authenticator Selection Criteria (dictionary
AuthenticatorSelectionCriteria)

5. Web Authentication API (2) (3) (4) (5) (6) (7)

1. Introduction (2) (3) (4) (5)

* 5.1. PublicKeyCredential Interface

5.1.3. Create a new credential - PublicKeyCredential's
sameOriginWithAncestors) method (2) (3) (4)

* 5.1.4. User about Public Credential (interface
AuthenticatorAttestationResponse)

PublicCredentialEntity (dictionary
AuthenticatorSelectionCriteria)

* 5.4.5. Authenticator Selection Criteria (dictionary
AuthenticatorSelectionCriteria)

3. Security Considerations

10.9. Biometric Authenticator Performance Bounds Extension
(biometricPerfBounds)
12.1. Registration (2) (3) (4) (5)
10.7. Location Extension (loc) (2)
10.4. Authenticator Selection Extension (authnSel) (2) (3)
9.6. Example Extension (dictionary)
9.4. Privacy
8.2. The authenticatorGetAssertion operation (2) (3)
6.3.5.2. Attestation Certificate and Attestation Certificate CA Compromise (2) (3) (4) (5) (6)
7. Relying Party Operations (2) (3) (4)
7.1. Registering a new credential (2) (3) (4) (5) (6) (7) (8) (9)
10. (11) (12) (13)
7.2. Verifying an authentication assertion (2) (3) (4) (5) (6) (7) (8)
8.4. Android Key Attestation Statement Format
9. WebAuthn Extensions (2) (3) (4) (5) (6)
9.2. Defining extensions (2)
9.3. Extending request parameters (2) (3) (4)
9.6. Example Extension (2) (3)
10.0. FIDO AppId Extension (appid) (2)
10.2. Simple Transaction Authorization Extension (txAuthSimple)
10.4. Authenticator Selection Extension (authnSel) (2) (3)
10.5. Supported Extensions Extension (exts) (2)
10.6. User Verification Index Extension (uvi)
10.7. Location Extension (loc) (2)
11.2. WebAuthn Extension Identifier Registrations (2)
12.1. Registration (2) (3) (4) (5)
12.2. Registration Specifically with User Verifying Platform Authentication (2) (3)
11.2. WebAuthn Extension Identifier Registrations

6.2.2. The authenticatorGetAssertion operation (2) (3) (4) (5)

6.2.1. The authenticatorMakeCredential operation (2) (3) (4) (5)

5. Web Authentication API

5.4. Options for Credential Creation (dictionary PublicKeyCredentialRequestOptions)

5.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)

4. Terminology

4.1. PublicKeyCredential

4.2. PublicKeyCredentialRequest

4.3. Transaction Authorization Extension (txAuth)

4.4. WebAuthn Authenticator

3. Attestation

2. FIO FIDO Alliance

1. Introduction

#relying-party-identifier

Referenced in:

5. Web Authentication API

5.1.3. Create a new credential - PublicKeyCredential's [Create](origin, options, sameOriginWithAncestors) method (2)

5.2.1. The authenticatorGetAssertion operation (2)

5.2.2. The authenticatorGetAssertion operation (2)

5.7.1. Registering a new credential (2)

5.7.2. Verifying an assertion (2)

5.8.8. Android Key Attestation Statement Format

5.8.6. FIDO U2F Attestation Statement Format

5.6. WebAuthn Authenticator model

5.1.4. Use an existing credential to make an assertion - PublicKeyCredential's [[Get](options) method

13. Security Considerations (2) (3) (4)

13.1. Cryptographic Challenges

13.2. Attestation Certificate and Attestation Certificate CA

Compromise (2) (3) (4) (5)

13.3. credentialUn Signed

14.1. Attestation Privacy

14.2. Registration Ceremony Privacy (2) (3) (4)

14.3. Authentication Ceremony Privacy (2) (3) (4)

#test-of-user-presence

Referenced in:

4. Terminology (2) (3) (4) (5) (6)

6.2.2. The authenticatorMakeCredential operation (2)

6.2.3. The authenticatorGetAssertion operation (2) (3)

5.2.3. The authenticatorGetAssertion operation (2)

7.1. Registering a new credential (2)

7.2. Verifying an assertion (2)

8. Android Key Attestation Statement Format

8.6. FIDO U2F Attestation Statement Format

10.1. FIDO AppID Extension (appid)

#test-of-user-presence

Referenced in:

4. Terminology (2) (3) (4) (5) (6)

6.2.2. The authenticatorMakeCredential operation (2)

6.2.3. The authenticatorGetAssertion operation (2)

10.2. Simple Transaction Authorization Extension (txAuthSimple)

10.3. Generic Transaction Authorization Extension (txAuthGeneric)

10.5. WebAuthn Authenticator model

1. Introduction (2)

4. Terminology (2)

5. Web Authentication API

5.1.3. Create a new credential - PublicKeyCredential's [[Create](origin, options, sameOriginWithAncestors) method (2) (3)

5.1.4. Use an existing credential to make an assertion - PublicKeyCredential's [[Get](options) method

5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource](origin, options, sameOriginWithAncestors) method (2)

5.4.2. RP Parameters for Credential Generation (dictionary PublicKeyCredentialRequestOptions)

6.1. Authenticator data (2) (3) (4) (5) (6)

6.1.1. Signature Counter Considerations

6.2.4. The authenticatorGetAssertion operation (2)

6.2.5. Attestation Conveyance Preference (enum)

6.2.6. The authenticatorGetAssertion operation (2)

10.4. Android Key Attestation Statement Format

10.5. FIDO U2F Attestation Statement Format

10.7. FIDO AppID Extension (appid)
6217
6218 6.2.2. The authenticatorGetAssertion operation
6219
6220 #user-handleReferenced in:
6221
6222 6.2.2. The authenticatorGetAssertion operation
6223
6224 #user-handleReferenced in:
6225
6226 6.2.2. The authenticatorGetAssertion operation
6227
6228 #user-handleReferenced in:
6229
6230 6.2.2. The authenticatorGetAssertion operation
6231
6232 #user-handleReferenced in:
6233
6234 6.2.2. The authenticatorGetAssertion operation
6235
6236 #user-handleReferenced in:
6237
6238 6.2.2. The authenticatorGetAssertion operation
6239
6240 #user-handleReferenced in:
6241
6242 6.2.2. The authenticatorGetAssertion operation
6243
6244 #user-handleReferenced in:
6245
6246 6.2.2. The authenticatorGetAssertion operation
6247
6248 #user-handleReferenced in:
6249
6250 6.2.2. The authenticatorGetAssertion operation
6251
6252 #user-handleReferenced in:
6253
6254 6.2.2. The authenticatorGetAssertion operation
6255
6256 #user-handleReferenced in:
6257
6258 6.2.2. The authenticatorGetAssertion operation
6259
6260 #user-handleReferenced in:
6261
6262 6.2.2. The authenticatorGetAssertion operation
6263
6264 #user-handleReferenced in:
6265
6266 6.2.2. The authenticatorGetAssertion operation
6267
6268 #user-handleReferenced in:
6269
6270 6.2.2. The authenticatorGetAssertion operation
6271
6272 #user-handleReferenced in:
6273
6274 6.2.2. The authenticatorGetAssertion operation
6275
6276 #user-handleReferenced in:
6277
6278 6.2.2. The authenticatorGetAssertion operation
6279
6280 #user-handleReferenced in:
6281
6282 6.2.2. The authenticatorGetAssertion operation
6283
6284 #user-handleReferenced in:
6285
6286 6.2.2. The authenticatorGetAssertion operation
6287
6288 #user-handleReferenced in:
6289
6290 6.2.2. The authenticatorGetAssertion operation
6291
6292 #user-handleReferenced in:
6293
6294 6.2.2. The authenticatorGetAssertion operation
6295
6296 #user-handleReferenced in:
6297
6298 6.2.2. The authenticatorGetAssertion operation
6299
6300 #user-handleReferenced in:
6301
6302 6.2.2. The authenticatorGetAssertion operation
6303
6304 #user-handleReferenced in:
6305
6306 6.2.2. The authenticatorGetAssertion operation
6307
6308 #user-handleReferenced in:
6309
6310 6.2.2. The authenticatorGetAssertion operation
6311
6312 #user-handleReferenced in:
6313
6314 6.2.2. The authenticatorGetAssertion operation
6315
6316 #user-handleReferenced in:
6317
6318 6.2.2. The authenticatorGetAssertion operation
6319
6320 #user-handleReferenced in:
6321
6322 6.2.2. The authenticatorGetAssertion operation
5.1. PublicKeyCredential Interface

5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2)

5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method (2)

5.1.5. Store an existing credential - PublicKeyCredential's [[Store]](credential, sameOriginWithAncestors) method (2)

5.1.6. Availability of User-Verifying Platform Authenticator - PublicKeyCredential's isUserVerifyingPlatformAuthenticatorAvailable() method

5.8.3. Credential Descriptor (dictionary PublicKeyCredentialDescriptor)

7.2. Verifying an authentication assertion

7.2. Verify an authentication assertion

7.2. Verify an authentication assertion

7.2. Verify an authentication assertion

7.2. Verify an authentication assertion

7.2. Verify an authentication assertion

7.2. Verify an authentication assertion

7.2. Verify an authentication assertion

7.2. Verify an authentication assertion

7.2. Verify an authentication assertion

7.2. Verify an authentication assertion

7.2. Verify an authentication assertion

7.2. Verify an authentication assertion
5.6. Abort operations with AbortSignal (2) (3) (4) (5)

5.6.2. The authenticatorMakeCredential operation

5.1.3. Create a new credential - PublicKeyCredential's

[[Create]](origin, options, sameOriginWithAncestors) method

5.1.4. Use an existing credential to make an assertion - PublicKeyCredential's

[[Get]](options) method

5.5. Abort operations with AbortSignal (2) (3) (4) (5)

5.6.2. The authenticatorMakeCredential operation

* 14.2. Registration Ceremony Privacy

* 14.3. Authenticator Attachment enumeration (enum AuthenticatorAttachment)

* 14.5. Authenticator Attachment enumeration (enum AuthenticatorAttachment)
5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method

#assertioncreationdata-signatureresultReferenced in:
5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method

#assertioncreationdata-userhandleresultReferenced in:
5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method

#assertioncreationdata-clientextensionresultsReferenced in:
5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method

#authenticatorresponseReferenced in:
5.1. PublicKeyCredential Interface
5.2. Authenticator Responses (interface AuthenticatorResponse)
5.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)
5.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)

#dom-authenticatorresponse-clientdatasonerferred in:
5.1.3. Create a new credential - PublicKeyCredential's
[[Create]](origin, options, sameOriginWithAncestors) method (2)
5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method (2)
5.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
5.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)
5.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)

#authenticatorestatementresponseReferenced in:
5.1. PublicKeyCredential Interface
5.1.3. Create a new credential - PublicKeyCredential's
[[Create]](origin, options, sameOriginWithAncestors) method
5.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)
5.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
5.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

#dom-authenticatorestatementresponse-attestationobjectReferenced in:
5.1.3. Create a new credential - PublicKeyCredential's
[[Create]](origin, options, sameOriginWithAncestors) method
5.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)
5.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

#assertioncreationresponseReferenced in:
5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method

#assertioncreationdata-signatureresultReferenced in:
5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method

#assertioncreationdata-userhandleresultReferenced in:
5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method

* 6.2.3. The authenticatorGetAssertion operation

* 7.1. Registering a new credential
* 7.2. Verifying an authentication assertion
* 7.1. Registering a new credential (2) (3)

#dom-authenticatorestatementresponse-attestationobjectReferenced in:
5.1.3. Create a new credential - PublicKeyCredential's
[[Create]](origin, options, sameOriginWithAncestors) method
5.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)
5.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

* 7.1. Registering a new credential
* 7.2. Verifying an authentication assertion

* 6.2. The authenticatorGetAssertion operation

* 7.1. Registering a new credential
* 7.2. Verifying an authentication assertion

* 7.2. Registering a new credential (2)

* 6.2.3. The authenticatorGetAssertion operation
#dom-makepublickeycredentialoptions-pubkeycredparams
Referenced in:

- 5.4. Options for Credential Creation (dictionary PublicKeyCredentialOptions)
- 5.1.1. CredentialCreationOptions Dictionary Extension
- 5.3. Parameters for Credential Generation (dictionary PublicKeyCredentialParameters)

#dictdef-publickeycredentialparametersReferenced in:

- 5.3. Parameters for Credential Generation (dictionary PublicKeyCredentialParameters)
- 5.4. Options for Credential Creation (dictionary PublicKeyCredentialCreationOptions)

#dom-publickeycredentialparameters-typeReferenced in:

- 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2)
- 5.3. Parameters for Credential Generation (dictionary PublicKeyCredentialParameters)

#dom-publickeycredentialparameters-algReferenced in:

- 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
- 5.3. Parameters for Credential Generation (dictionary PublicKeyCredentialParameters)

#dictdef-makepublickeycredentialoptionsReferenced in:

- 5.1.1. CredentialCreationOptions Extension
- 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
- 5.4. Options for Credential Creation (dictionary PublicKeyCredentialCreationOptions)

#dom-makepublickeycredentialoptions-propsReferenced in:

- 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
- 5.4. Options for Credential Creation (dictionary PublicKeyCredentialCreationOptions)

#dom-makepublickeycredentialoptions-userReferenced in:

- 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
- 5.4. Options for Credential Creation (dictionary PublicKeyCredentialCreationOptions)

- 7.1. Registering a new credential

#dom-makepublickeycredentialoptions-challengeReferenced in:

- 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
- 5.4. Options for Credential Creation (dictionary PublicKeyCredentialCreationOptions)

#dictdef-makepublickeycredentialoptionsReferenced in:

- 5.1.1. CredentialCreationOptions Extension
- 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
- 5.4. Options for Credential Creation (dictionary PublicKeyCredentialCreationOptions)

#dom-publickeycredentialcreationoptions-rpReferenced in:

- 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
- 5.4. Options for Credential Creation (dictionary PublicKeyCredentialCreationOptions)

#dom-publickeycredentialcreationoptions-userReferenced in:

- 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
- 5.4. Options for Credential Creation (dictionary PublicKeyCredentialCreationOptions)

- 7.1. Registering a new credential

#dom-publickeycredentialcreationoptions-challengeReferenced in:

- 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
- 5.4. Options for Credential Creation (dictionary PublicKeyCredentialCreationOptions)

- 13.1. Cryptographic Challenges

#dom-makepublickeycredentialoptions-pubkeycredparamsReferenced in:
6.2.1. The authenticatorMakeCredential operation (2)

**5.4. Options for Credential Creation (dictionary PublicKeyCredentialEntity)**

- 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2)

**5.4. Options for Credential Creation (dictionary PublicKeyCredentialCreationOptions)**

- 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2)

**5.4. Options for Credential Creation (dictionary PublicKeyCredentialCreationOptions)**

- 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2)

**5.4. Options for Credential Creation (dictionary PublicKeyCredentialCreationOptions)**

- 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2)

**5.4. Options for Credential Creation (dictionary PublicKeyCredentialCreationOptions)**

- 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2)

**5.4. Options for Credential Creation (dictionary PublicKeyCredentialCreationOptions)**
5.4. Options for Credential Creation (dictionary PublicKeyCredentialOptions) (2)

6.2. The authenticatorMakeCredential operation (2) (3)

#dictdef-publickeycredentialuserentity-displaynameReferenced in:
* 5.4.3. User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity) (2)
* 6.2. The authenticatorMakeCredential operation (2) (3)
#platform-authenticatorsReferenced in:
* 5.1.6. Availability of User-Verifying Platform Authenticator - PublicKeyCredential's
isUserVerifyingPlatformAuthenticatorAvailable() method (2) (3) (4)

#platform-authenticatorsReferenced in:
* 5.4.5. Authenticator Attachment enumeration (enum AuthenticatorAttachment) (2)

#roaming-authenticatorsReferenced in:
* 1.1.3. Other use cases and configurations

#cross-platform-attachedReferenced in:
* 5.4.5. Authenticator Attachment enumeration (enum AuthenticatorAttachment) (2)

#attestation-conveyanceReferenced in:
* 5.4. Options for Credential Creation (dictionary PublicKeyCredentialCreationOptions)
* 5.4.6. Attestation Conveyance Preference enumeration (enum AttestationConveyancePreference)

#enumdef-attestationconveyancepreferenceReferenced in:
* 5.4.6. Attestation Conveyance Preference enumeration (enum AttestationConveyancePreference)

#dom-attestationconveyancepreference-noneReferenced in:
* 5.4. Options for Credential Creation (dictionary PublicKeyCredentialCreationOptions)
* 5.4.6. Attestation Conveyance Preference enumeration (enum AttestationConveyancePreference)

#dom-attestationconveyancepreference-indirectReferenced in:
* 5.4.6. Attestation Conveyance Preference enumeration (enum AttestationConveyancePreference)

#dom-attestationconveyancepreference-directReferenced in:
* 5.4.6. Attestation Conveyance Preference enumeration (enum AttestationConveyancePreference)

#dictdef-publickeycredentialrequestoptionsReferenced in:
* 5.1.2. CredentialRequestOptions Dictionary Extension
* 5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource]Origin, options, sameOriginWithAncestors] method

* 5.5. Options for Assertion Generation (dictionary
* 7.2. Verifying an authentication assertion

#dom-publickeycredentialrequestoptions-challengeReferenced in:
* 5.1.4.1. PublicKeyCredential's
  [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method
* 5.5. Options for Assertion Generation (dictionary
  PublicKeyCredentialRequestOptions)
* 13.1. Cryptographic Challenges

#dom-publickeycredentialrequestoptions-timeoutReferenced in:
* 5.1.4.1. PublicKeyCredential's
  [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method (2)
* 5.5. Options for Assertion Generation (dictionary
  PublicKeyCredentialRequestOptions)

#dom-publickeycredentialrequestoptions-rpidReferenced in:
* 5.1.4.1. PublicKeyCredential's
  [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method (2) (3) (4)
* 5.5. Options for Assertion Generation (dictionary
  PublicKeyCredentialRequestOptions)

#dom-publickeycredentialrequestoptions-allowcredentialsReferenced in:
* 5.1.4.1. PublicKeyCredential's
  [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method (2) (3) (4)
* 5.5. Options for Assertion Generation (dictionary
  PublicKeyCredentialRequestOptions)

#dictdef-authenticationextensionsclientinputsReferenced in:
* 5.4. Options for Credential Creation (dictionary
  PublicKeyCredentialCreationOptions)
* 5.5. Options for Assertion Generation (dictionary
  PublicKeyCredentialRequestOptions)

#dictdef-authenticationextensionsclientinputsReferenced in:
* 5.1. PublicKeyCredential Interface
  [[Create]](origin, options, sameOriginWithAncestors) method
* 5.1.4.1. PublicKeyCredential's
  [[DiscoverFromExternalSource]](origin, options,
5.1.4.1. PublicKeyCredential's #dom-collectedclientdata-hashalgorithm
Referenced in:
7.2. Verifying an authentication assertion
7.1. Registering a new credential

5.1.4.1. PublicKeyCredential's #dom-collectedclientdata-origin
Referenced in:
7.1. Registering a new credential

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

5.8.1. Client data used in WebAuthn signatures (dictionary CollectedClientData)
5.1.3. Create a new credential - PublicKeyCredential's #dom-collectedclientdata-origin
5.1.3. Create a new credential - PublicKeyCredential's CollectedClientData)

CollectedClientData)

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

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

5.2. Authenticator Responses (interface AuthenticatorResponse)
6. WebAuthn Authenticator model (2) (3) (4)
7.1. Registering a new credential
7.2. Verifying an authentication assertion
9. WebAuthn Extensions

5.4. Options for Credential Creation (dictionary MakePublicKeyCredentialOptions) (2)

5.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions) (2)

5.3. Create a new credential - PublicKeyCredential's #dom-collectedclientdata-type
Referenced in:
7.2. Verifying an authentication assertion
7.1. Registering a new credential

5.1.3. Create a new credential - PublicKeyCredential's #dictdef-collectedclientdata
Referenced in:

5.1.3. Create a new credential - PublicKeyCredential's #dictdef-collectedclientdata
7.1. Registering a new credential

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

7.2. Verifying an authentication assertion
7.1. Registering a new credential

1.7.1. Registering a new credential
1.7.2. Verifying an authentication assertion

1.7.1. Registering a new credential
1.7.2. Verifying an authentication assertion

9. WebAuthn Extensions

7.1. Registering a new credential
7.2. Verifying an authentication assertion

10.7. Location Extension (loc)
10.6. User Verification Index Extension (uvn)
10.5. Supported Extensions Extension (exts)
10.4. Authenticator Selection Extension (authnSel)
10.3. Generic Transaction Authorization Extension (txAuthGeneric)
10.2. Simple Transaction Authorization Extension (txAuthSimple)
10.1. FIDO AppID Extension (appid)
9. WebAuthn Extensions

10.2. Simple Transaction Authorization Extension (txAuthSimple)
10.1. FIDO AppID Extension (appid)
9. WebAuthn Extensions

2. Verifying an authentication assertion
1. Registering a new credential
6.2.1. The authenticatorMakeCredential operation

5.8.1. Client data used in WebAuthn signatures (dictionary CollectedClientData) (2)

7.1. Registering a new credential
7.2. Verifying an authentication assertion

#collectedclientdata-tokenbindingReferenced in:
5.1.3. Create a new credential - PublicKeyCredential's (2)
5.1.4.1. PublicKeyCredential's (2)
5.8.1. Client data used in WebAuthn signatures (dictionary CollectedClientData)
7.1. Registering a new credential
7.2. Verifying an authentication assertion

9.4. Client extension processing

#collectedclientdata-authenticatorextensionsReferenced in:
5.1.3. Create a new credential - PublicKeyCredential's (2)
5.1.4.1. PublicKeyCredential's (2)
5.8.1. Client data used in WebAuthn signatures (dictionary CollectedClientData)
7.1. Registering a new credential
7.2. Verifying an authentication assertion

#collectedclientdata-json-serialized-client-dataReferenced in:
5.1.3. Create a new credential - PublicKeyCredential's (2)
5.1.4.1. PublicKeyCredential's (2)
5.8.1. Client data used in WebAuthn signatures (dictionary CollectedClientData)
7.1. Registering a new credential
7.2. Verifying an authentication assertion

#collectedclientdata-hash-of-the-serialized-client-dataReferenced in:
5.1.3. Create a new credential - PublicKeyCredential's (2)
5.1.4.1. PublicKeyCredential's (2)
5.8.1. Client data used in WebAuthn signatures (dictionary CollectedClientData)
6.2.1. The authenticatorMakeCredential operation
6.2.2. The authenticatorGetAssertion operation (2)


# 5.4. Authenticator Transport enumeration (enum AuthenticatorTransport)

- 5.4.1. AuthenticatorTransport-ble
- 5.4.2. AuthenticatorTransport-nfc
- 5.4.3. AuthenticatorTransport-web
- 5.4.4. AuthenticatorTransport-usb

# 6. WebAuthn Authenticator Model (2) (3)

- 6.1. Terminology
- 6.2. The authenticatorMakeCredential operation
- 6.3.1. Attested credential data
- 6.3.2. The authenticatorGetAssertion operation
- 6.4. WebAuthn Applicability
- 6.5. The authenticatorAttestationExtension
- 6.6. Authenticator Registration
- 6.7. AttestedCredentialData
- 6.8. WebAuthn Parameters
- 6.9. Attestation

**#enumdef-authenticatorcredentials Referenced in:**
  - 6.2.1. The authenticatorMakeCredential operation
  - 6.2.2. The authenticatorGetAssertion operation

**#typedefdef-cosealgorithmidentifier Referenced in:**
  - 5.8.5. Cryptographic Algorithm Identifier (typedef COSEAlgorithmIdentifier)

**#dom-authenticatortransport-ble Referenced in:**
  - 5.8.4. Authenticator Transport enumeration (enum AuthenticatorTransport)

**#dom-authenticatortransport-nfc Referenced in:**
  - 5.8.4. Authenticator Transport enumeration (enum AuthenticatorTransport)

**#enumdef-userverificationrequirement Referenced in:**
  - 4. Terminology
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
  - 5.1.1.4.1. PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method

**#typedefdef-cosealgorithmidentifier Referenced in:**
  - 5.10.4. Authenticator Transport enumeration (enum AuthenticatorTransport)

**#enumdef-userverificationrequirement Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#enumdef-userverificationrequirement Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#enumdef-userverificationrequirement Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#enumdef-userverificationrequirement Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

**#typedefdef-causealgorithmidentifier Referenced in:**
  - 5.10.6. User Verification Requirement enumeration (enum UserVerificationRequirement)
6.1. Authenticator data

6.2.2. The authenticatorGetAssertion operation (2) (3)

6.1. Authenticator data (2)

6.2.2. The authenticatorMakeCredential operation

6.2.3. The authenticatorGetAssertion operation

6.3. Attestation (2)

6.3.1. Attested credential data

6.3.2. Attestation Statement Formats (2)

6.3.4. Generating an Attestation Object

6.3.5.3. Attestation Certificate Hierarchy

6.4. Example Extension (2)

6.10.6. User Verification Method Extension (uvm)

10.6. User Verification Index Extension (uvl)

7.2. Verifying an authentication assertion

7.2. Verifying an authentication assertion (2) (3)

7.1. Registering a new credential

7.2. Verifying an authentication assertion (2) (3)

7.1. Registering a new credential (2)

8.3. TPM Attestation Statement Format

8.4. Android Key Attestation Statement Format

8.6. FIDO U2F Attestation Statement Format

8.6.1. Signing a new credential

8.6.1. Signing a new credential (2)

8.6.1. Signing a new credential (2) (3)

8.6. FIDO U2F Attestation Statement Format

8.6.2. The authenticatorMakeCredential operation

8.6.3. The authenticatorGetAssertion operation

8.6.4. Signing a new credential (2)

8.6.5. Signing a new credential (2) (3)

8.6.6. Signing a new credential (2) (3) (4)

8.6.7. Signing a new credential (2) (3) (4) (5)

8.6.8. Signing a new credential (2) (3) (4) (5) (6)

8.6.9. Signing a new credential (2) (3) (4) (5) (6) (7)

8.6.10. Signing a new credential (2) (3) (4) (5) (6) (7) (8)

8.6.11. Signing a new credential (2) (3) (4) (5) (6) (7) (8) (9)

8.6.12. Signing a new credential (2) (3) (4) (5) (6) (7) (8) (9) (10)


8.6.15. Signing a new credential (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13)


8.3. TPM Attestation Statement Format

#attestation-statement-format
Referenced in:
- 6.3.4. Generating an Attestation Object
- 6.3.2. Attestation Statement Formats (2) (3) (4) (5) (6) (7) (8)
- 7.1. Registering a new credential
- 5.10.4. Authenticator Transport enumeration (enum AuthenticatorTransport)
- 5.2.1. Information about Public Key Credential (interface PublicKeyCredential's [Create](origin, options, sameOriginWithAncestors) method

7.1. Registering a new credential

#attestation-statement-format
Referenced in:
- 6.3.4. Generating an Attestation Object
- 6.3.2. Attestation Statement Formats (2) (3) (4) (5) (6) (7) (8)
- 7.1. Registering a new credential
- 5.10.4. Authenticator Transport enumeration (enum AuthenticatorTransport)
- 5.2.1. Information about Public Key Credential (interface PublicKeyCredential's [Create](origin, options, sameOriginWithAncestors) method

7.1. Registering a new credential
Compromise

6.3.5.2. Attestation Certificate and Attestation Certificate CA

\#ecdaa

Referenced in:
1. 6.3.5.1. Privacy

\#elliptic-curve-based-direct-anonymous-attestation

Referenced in:
1. 8.6. FIDO U2F Attestation Statement Format

\#authenticator-data-claimed-to-have-been-used-for-the-attestation

nced in:
1. * 8.4. Android Key Attestation Statement Format

\#attestation-trust-path

Referenced in:
1. 6.3.2. Attestation Statement Formats

2. 6.3.2. Attestation Statement Formats (2) (3)

3. 8.3. TPM Attestation Statement Format

4. 8.4. Android Key Attestation Statement Format

5. 8.5. Android SafetyNet Attestation Statement Format

6. 8.6. FIDO U2F Attestation Statement Format

\#basic-attestation

Referenced in:
1. * 6.3.5.1. Privacy

2. * 6.4. Android Key Attestation Statement Format

3. * 8.4. Android Key Attestation Statement Format


5. * 8.6. FIDO U2F Attestation Statement Format

\#self-attestation

Referenced in:
1. * 4. Terminology (2) (3) (4)

2. * 5.4.6. Attestation Conveyance Preference enumeration (enum

3. AttestationConveyancePreference)

4. * 6.3. Attestation (2)

5. * 6.3.2. Attestation Statement Formats

6. * 6.3.3. Attestation Types

7. * 6.3.5.2. Attestation Certificate and Attestation Certificate CA

8. * 7.1. Registering a new credential (2) (3)

9. * 8.2. Packed Attestation Statement Format (2)


11. * 8.6. FIDO U2F Attestation Statement Format

\#privacy-ca

Referenced in:
1. * 5.1.3. Create a new credential - PublicKeyCredential's

2. [[Create](origin, options, sameOriginWithAncestors)] method

3. * 5.4.6. Attestation Conveyance Preference enumeration (enum

4. AttestationConveyancePreference)

5. * 6.3.5.1. Privacy

6. * 8.5. TPM Attestation Statement Format

\#elliptic-curve-based-direct-anonymous-attestation

Referenced in:
1. * 6.3.5.1. Privacy

2. * 6.3.2. Attestation Statement Formats

3. * 6.3.3. Attestation Types

4. * 6.3.5.2. Attestation Certificate and Attestation Certificate CA

5. Compromise

\#ecdaa

Referenced in:
1. * 6.3.2. Attestation Statement Formats

2. * 6.3.3. Attestation Types

3. * 6.3.5.2. Attestation Certificate and Attestation Certificate CA

Compromise
7.4. Client extension processing

9.4.2. Defining extensions

9. WebAuthn Extensions

11.2. WebAuthn Extension Identifier Registrations

10.6. User Verification Index Extension (uvi)

10.8. User Verification Method Extension (uvm)

11.2. WebAuthn Extension Identifier Registrations

#registration-extensionReferenced in:
#registration-extensionReferenced in:
#registration-extensionReferenced in:

#authentication-extensionReferenced in:
#authentication-extensionReferenced in:
#client-extensionReferenced in:
#client-extensionReferenced in:
#authenticator-extensionReferenced in:
#client-extension-outputReferenced in:
7.2. Verifying an authentication assertion (2)
7.1. Registering a new credential (2)
9. WebAuthn Extensions (2)
9.2. Defining extensions (2)
9.5. Authenticator extension processing (2)
10.5. Supported Extensions Extension (exts) (2)
10.7. Location Extension (loc)
11.2. WebAuthn Extension Identifier Registrations

#extension-identifierReferenced in:
9.5. Authenticator extension processing (2)
9.3. Extending request parameters (2)
9.4. Client extension processing (2)
9.5. Authenticator extension processing (2)
6.2.2. The authenticatorMakeCredential operation (2)
6.2.3. The authenticatorGetAssertion operation (2)
6.2.1. The authenticatorMakeCredential operation (2)

* 9. WebAuthn Extensions (2)
* 9.2. Defining extensions (2)
* 9.3. Extending request parameters (2)
* 9.4. Client extension processing (2)
* 9.5. Authenticator extension processing (2)

#extension-identifierReferenced in:
5.9. Authentication Extensions Authenticator Inputs (typedef AuthenticationExtensionsClientInputs)
7.1. Registering a new credential (2)
7.2. Verifying an authentication assertion (2)
9. WebAuthn Extensions (2)
9.2. Defining extensions (2)
9.3. Extending request parameters (2)
9.4. Client extension processing (2)
9.5. Authenticator extension processing (2)

* 9. WebAuthn Extensions (2)
* 9.2. Defining extensions (2)
* 9.3. Extending request parameters (2)
* 9.4. Client extension processing (2)
* 9.5. Authenticator extension processing (2)

#client-extension-inputReferenced in:
5.7. Authentication Extensions Client Inputs (typedef AuthenticationExtensionsClientInputs)
7.1. Registering a new credential (2)
7.2. Verifying an authentication assertion (2)
9. WebAuthn Extensions (2)
9.2. Defining extensions (2)
9.3. Extending request parameters (2)
9.4. Client extension processing (2)
9.5. Authenticator extension processing (2)

* 9. WebAuthn Extensions (2)
* 9.2. Defining extensions (2)
* 9.3. Extending request parameters (2)
* 9.4. Client extension processing (2)
* 9.5. Authenticator extension processing (2)

#client-extension-outputReferenced in:
5.1. PublicKeyCredential Interface
5.1.3. Create a new credential - PublicKeyCredential's
[[Create](origin, options, sameOriginWithAncestors) method (2)
5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource](origin, options, sameOriginWithAncestors) method (2)
5.1.3. Create a new credential - PublicKeyCredential's
[[Create](origin, options, sameOriginWithAncestors) method (2)
5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource](origin, options, sameOriginWithAncestors) method (2)
5.1.3. Create a new credential - PublicKeyCredential's
[[Create](origin, options, sameOriginWithAncestors) method (2)
5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource](origin, options, sameOriginWithAncestors) method (2)
5.1.3. Create a new credential - PublicKeyCredential's
[[Create](origin, options, sameOriginWithAncestors) method (2)
5.1.4.1. PublicKeyCredential's
10.4. Authenticator Selection Extension (authnSel)

#typedefdef-aaguid

Referenced in:

9. WebAuthn Extensions (2) (3)

9.4. Client extension processing (2) (3)

9.6. Example Extension

#typedefdef-authenticatorselectionlist

Referenced in:

authenticator-extension-processing

Referenced in:

authenticator-extension-output

Referenced in:

9. WebAuthn Extensions (2) (3)

9.1. Authenticator data

9.6. Example Extension

10.5. Supported Extensions Extension (exts)

10.5. Authenticator extension processing

10.6. Example Extension

10.5. Supported Extensions Extension (exts)

10.6. User Verification Index Extension (uvi)

10.7. Location Extension (loc)

10.8. User Verification Method Extension (uvm)

#typedefdef-authenticatorextensionssupported

Referenced in:

10.4. Authenticator Selection Extension (authnSel)

#typedefdef-aaguid

Referenced in:

10.4. Authenticator Selection Extension (authnSel)

#typedefdef-auuidReference

Referenced in:

5.4.6. Attestation Conveyance Preference enumeration (enum)

AttestationConveyancePreference

Referenced in:

1.4.1. Attestation Privacy (2) (3)