THE_URL:file://localhost/Users/jehodges/documents/work/standards/W3C/webauthn/index-master-tr-5e63e57-WD-07.html
THE_TITLE:Web Authentication: An API for accessing Public Key Credentials - Level 1
^| Jump to Table of Contents-> Pop Out Sidebar

W3C

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

W3C Working Draft, 5 December 2017

This version:
https://www.w3.org/TR/2017/WD-webauthn-20171205/

Latest published Version:
https://www.w3.org/TR/webauthn/

Editor's Draft:
https://w3c.github.io/webauthn/

Previous versions:

https://www.w3.org/TR/2017/WD-webauthn-20170811/
https://www.w3.org/TR/2017/WD-webauthn-20170505/
https://www.w3.org/TR/2017/WD-webauthn-20170216/
https://www.w3.org/TR/2016/WD-webauthn-20161207/
https://www.w3.org/TR/2016/WD-webauthn-20160928/
https://www.w3.org/TR/2016/WD-webauthn-20160902/
https://www.w3.org/TR/2016/WD-webauthn-20160531/

Issue Tracking:
Github

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)
J.C. Jones (Mozilla)

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

Abstract

This specification defines an API enabling the creation and use of

THE_URL:file://localhost/Users/jehodges/documents/work/standards/W3C/webauthn/index-master-tr-716a169-WD-08a.html
THE_TITLE:Web Authentication: An API for accessing Public Key Credentials Level 1
^| Jump to Table of Contents-> Pop Out Sidebar

W3C

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

W3C Working Draft, 6 March 2018

This version:
https://www.w3.org/TR/2018/WD-webauthn-20180306/

Latest published version:
https://www.w3.org/TR/webauthn/

Editor's Draft:
https://w3c.github.io/webauthn/

Previous Versions:
https://www.w3.org/TR/2017/WD-webauthn-20171205/
https://www.w3.org/TR/2017/WD-webauthn-20170811/
https://www.w3.org/TR/2017/WD-webauthn-20170505/
https://www.w3.org/TR/2017/WD-webauthn-20170216/
https://www.w3.org/TR/2016/WD-webauthn-20161207/
https://www.w3.org/TR/2016/WD-webauthn-20160928/
https://www.w3.org/TR/2016/WD-webauthn-20160902/
https://www.w3.org/TR/2016/WD-webauthn-20160531/

Issue Tracking:
GitHub

Editors:

Dirk Balfanz (Google)
Alexei Czeskis (Google)

Jeff Hodges (PayPal)
J.C. Jones (Mozilla)
Michael B. Jones (Microsoft)
Akshay Kumar (Microsoft)
Angelo Liao (Microsoft)
Rolf Lindemann (Nok Nok Labs)
Emil Lundberg (Yubico)

Former Editors:
Vijay Bharadwaj (Microsoft)
Arnar Birgisson (Google)
Hubert Le Van Gong (PayPal)

Contributors:
Christiaan Brand (Google)
Adam Langley (Google)
Giridhar Mandyam (Qualcomm)
Mike West (Google)
Johan Verrept (VASCO Data Security)
Jeffrey Yasskin (Google)

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

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

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 February 2018 W3C Process Document.

Table of Contents

/Users/jehodges/Documents/work/standards/W3C/webauthn/index-master-tr-5e63e57-WD-07.txt, Top line: 121

PublicKeyCredential's [[Store]](credential, sameOriginWithAncestors) method
6. 5.1.6 Availability of User-Verifying Platform
Authenticator - PublicKeyCredential's isUserVerifyingPlatformAuthenticatorAvailable() method
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)
4. 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 UserVerificationRequirement)
6. 6 WebAuthn Authenticator model
1. 6.1 Authenticator data
1. 6.1.1 Signature Counter Considerations
2. 6.2 Authenticator operations
1. 6.2.1 The authenticatorMakeCredential operation
2. 6.2.2 The authenticatorGetAssertion operation
3. 6.2.3 The authenticatorCancel operation
3. 6.3 Attestation
1. 6.3.1 Attested credential data
2. 6.3.2 Attestation Statement Formats
3. 6.3.3 Attestation Types
4. 6.3.4 Generating an Attestation Object
5. 6.3.5 Security Considerations
1. 6.3.5.1 Privacy
2. 6.3.5.2 Attestation Certificate and Attestation

/Users/jehodges/Documents/work/standards/W3C/webauthn/index-master-tr-716a169-WD-08a.txt, Top line: 137

PublicKeyCredential's [[Store]](credential, sameOriginWithAncestors) method
6. 5.1.6 Preventing silent access to an existing credential - PublicKeyCredential's [[preventSilentAccess]](credential, sameOriginWithAncestors) method
7. 5.1.7 Availability of User-Verifying Platform
Authenticator - PublicKeyCredential's isUserVerifyingPlatformAuthenticatorAvailable() method
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)
4. 5.4 Options for Credential Creation (dictionary PublicKeyCredentialCreationOptions)
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 Client Inputs (typedef AuthenticationExtensionsClientInputs)
8. 5.8 Authentication Extensions Client Outputs (typedef AuthenticationExtensionsClientOutputs)
9. 5.9 Authentication Extensions Authenticator Inputs (typedef AuthenticationExtensionsAuthenticatorInputs)
10. 5.10 Supporting Data Structures
1. 5.10.1 Client data used in WebAuthn signatures (dictionary CollectedClientData)
2. 5.10.2 Credential Type enumeration (enum PublicKeyCredentialType)
3. 5.10.3 Credential Descriptor (dictionary PublicKeyCredentialDescriptor)
4. 5.10.4 Authenticator Transport enumeration (enum AuthenticatorTransport)
5. 5.10.5 Cryptographic Algorithm Identifier (typedef COSEAlgorithmIdentifier)
6. 5.10.6 User Verification Requirement enumeration (enum UserVerificationRequirement)
6. 6 WebAuthn Authenticator Model
1. 6.1 Authenticator data
1. 6.1.1 Signature Counter Considerations
2. 6.2 Authenticator operations
1. 6.2.1 Lookup Credential Source by Credential ID algorithm
2. 6.2.2 The authenticatorMakeCredential operation
3. 6.2.3 The authenticatorGetAssertion operation
4. 6.2.4 The authenticatorCancel operation
3. 6.3 Attestation
1. 6.3.1 Attested credential data
1. 6.3.1.1 Examples of credentialPublicKey Values encoded in COSE_Key format
2. 6.3.2 Attestation Statement Formats
3. 6.3.3 Attestation Types
4. 6.3.4 Generating an Attestation Object
5. 6.3.5 Signature Formats for Packed Attestation, FIDO U2F Attestation, and Assertion Signatures

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.

Relying Parties employ the Web Authentication API during two distinct, but related, ceremonies involving a user. The first is Registration, where a public key credential is created on an authenticator, and associated by a Relying Party with the present user's account (the account may already exist or may be created at this time). The second is Authentication, where the Relying Party is presented with an Authentication Assertion proving the presence and consent of the user who registered the public key credential. Functionally, the Web Authentication API comprises a PublicKeyCredential which extends the Credential Management API [CREDENTIAL-MANAGEMENT-1], and infrastructure which allows those credentials to be used with navigator.credentials.create() and navigator.credentials.get(). The former is used during Registration, and the latter during Authentication.

Broadly, compliant authenticators protect public key credentials, and interact with user agents to implement the Web Authentication API. Some authenticators may run on the same computing device (e.g., smart phone, tablet, desktop PC) as the user agent is running on. For instance, such an authenticator might consist of a Trusted Execution Environment (TEE) applet, a Trusted Platform Module (TPM), or a Secure Element (SE) integrated into the computing device in conjunction with some means for user verification, along with appropriate platform software to mediate access to these components' functionality. Other authenticators may operate autonomously from the computing device running the user agent, and be accessed over a transport such as Universal Serial Bus (USB), Bluetooth Low Energy (BLE) or Near Field Communications (NFC).

1.1. Use Cases

The below use case scenarios illustrate use of two very different types of authenticators, as well as outline further scenarios. Additional scenarios, including sample code, are given later in 12 Sample scenarios.

1.1.1. Registration

* On a phone:
  + User navigates to example.com in a browser and signs in to an existing account using whatever method they have been using (possibly a legacy method such as a password), or creates a new account.
  + The phone prompts, "Do you want to register this device with example.com?"
  + User agrees.
  + The phone prompts the user for a previously configured authorization gesture (PIN, biometric, etc.); the user provides this.
  + Website shows message, "Registration complete."

1.1.2. Authentication

* On a laptop or desktop:
  + User navigates to example.com in a browser, sees an option to "Sign in with your phone."
  + User chooses this option and gets a message from the browser,

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.

Relying Parties employ the Web Authentication API during two distinct, but related, ceremonies involving a user. The first is Registration, where a public key credential is created on an authenticator, and associated by a Relying Party with the present user's account (the account MAY already exist or MAY be created at this time). The second is Authentication, where the Relying Party is presented with an Authentication Assertion proving the presence and consent of the user who registered the public key credential. Functionally, the Web Authentication API comprises a PublicKeyCredential which extends the Credential Management API [CREDENTIAL-MANAGEMENT-1], and infrastructure which allows those credentials to be used with navigator.credentials.create() and navigator.credentials.get(). The former is used during Registration, and the latter during Authentication.

Broadly, compliant authenticators protect public key credentials, and interact with user agents to implement the Web Authentication API. Some authenticators MAY run on the same computing device (e.g., smart phone, tablet, desktop PC) as the user agent is running on. For instance, such an authenticator might consist of a Trusted Execution Environment (TEE) applet, a Trusted Platform Module (TPM), or a Secure Element (SE) integrated into the computing device in conjunction with some means for user verification, along with appropriate platform software to mediate access to these components' functionality. Other authenticators MAY operate autonomously from the computing device running the user agent, and be accessed over a transport such as Universal Serial Bus (USB), Bluetooth Low Energy (BLE) or Near Field Communications (NFC).

1.1. Use Cases

The below use case scenarios illustrate use of two very different types of authenticators, as well as outline further scenarios. Additional scenarios, including sample code, are given later in 12 Sample scenarios.

1.1.1. Registration

* On a phone:
  + User navigates to example.com in a browser and signs in to an existing account using whatever method they have been using (possibly a legacy method such as a password), or creates a new account.
  + The phone prompts, "Do you want to register this device with example.com?"
  + User agrees.
  + The phone prompts the user for a previously configured authorization gesture (PIN, biometric, etc.); the user provides this.
  + Website shows message, "Registration complete."

1.1.2. Authentication

* On a laptop or desktop:
  + User navigates to example.com in a browser, sees an option to "Sign in with your phone."
  + User chooses this option and gets a message from the browser,

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

1.1.3. Other use cases and configurations

A variety of additional use cases and configurations are also possible, including (but not limited to):
* A user navigates to example.com on their laptop, is guided through a flow to create and register a credential on their phone.
* A user obtains an discrete, roaming authenticator, such as a "fob" with USB or USB+NFC/BLE connectivity options, loads example.com in their browser on a laptop or phone, and is guided though a flow to create and register a credential on the fob.
* A Relying Party prompts the user for their authorization gesture in order to authorize a single transaction, such as a payment or other financial transaction.

2. Conformance

This specification defines three conformance classes. Each of these classes is specified so that conforming members of the class are secure against non-conforming or hostile members of the other classes.

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 end 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.

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.

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

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 though a flow to create and register a credential on the fob.
* A Relying Party prompts the user for their authorization gesture in order to authorize a single transaction, such as a payment or other financial transaction.

2. Conformance

This specification defines three conformance classes. Each of these classes is specified so that conforming members of the class are secure against non-conforming or hostile members of the other classes.

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 end 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.

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.

2.2.1. Backwards Compatibility with FIDO U2F

Authenticators that only support the 8.6 FIDO U2F Attestation Statement Format have no mechanism to store a user handle, so the returned userHandle will always be null.

2.3. Relying Parties

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

2.4. All Conformance Classes

All CBOR encoding performed by the members of the above conformance

3. Dependencies

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

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.

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

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

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

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

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

ECMAScript
: %ArrayBuffer% is defined in [ECMAScript].

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

Web IDL
: Many of the interface definitions and all of the IDL in this specification depend on [WebIDL-1]. This updated version of the Web IDL standard adds support for Promises, which are now the preferred mechanism for asynchronous interaction in all new web APIs.

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

4. Terminology

Assertion
: See Authentication Assertion.

classes MUST be done using the CTAP2 canonical CBOR encoding form. All decoders of the above conformance classes SHOULD reject CBOR that is not validly encoded in the CTAP2 canonical CBOR encoding form and SHOULD reject messages with duplicate map keys.

3. Dependencies

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

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.

CBOR
: A number of structures in this specification, including attestation statements and extensions, are encoded using the CTAP2 canonical CBOR encoding form of the Compact Binary Object Representation (CBOR) [RFC7049], as defined in [FIDO-CTAP].

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

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

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

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

ECMAScript
: %ArrayBuffer% is defined in [ECMAScript].

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

Web IDL
: Many of the interface definitions and all of the IDL in this specification depend on [WebIDL-1]. This updated version of the Web IDL standard adds support for Promises, which are now the preferred mechanism for asynchronous interaction in all new web APIs.

FIDO AppID
: The algorithms for determining the FacetID of a calling application and determining if a caller's FacetID is authorized for an AppID (used only in the appid extension) are defined by [FIDO-APPID].

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

Assertion
: See Authentication Assertion.

Attestation
Generally, attestation is a statement serving to bear witness, confirm, or authenticate. In the WebAuthn context, attestation is employed to attest to the provenance of an authenticator and the data it emits; including, for example: credential IDs, credential key pairs, signature counters, etc. 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.

Authentication
The ceremony where a user, and the user's computing device(s) (containing at least one authenticator) work in concert to cryptographically prove to an Relying Party that the user controls the 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.

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

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 Authentication Assertion, a challenge and other data presented by a Relying Party (in concert with the WebAuthn Client).

Authorization Gesture
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].

Biometric Authenticator
Any authenticator that implements biometric recognition.

Ceremony
The concept of a ceremony [Ceremony] is an extension of the concept of a network protocol, with human nodes alongside

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

Client
: See Conforming User Agent.

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 cases 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.

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

  Credential 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.

Credential Public Key
: The public key portion of an 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.

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

Client
: See WebAuthn Client, 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 cases 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.

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

  Credential 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.

Credential Public Key
: The public key portion of a Relying Party-specific credential key pair, generated by an authenticator and returned to a 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.

Human Palatability
: An identifier that is human-palatable is intended to be rememberable and reproducible by typical human users, in contrast to identifiers that are, for example, randomly generated sequences of bits [EduPersonObjectClassSpec].

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 displayName.

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"
Credential 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

Public Key Credential Source
A credential source ([CREDENTIAL-MANAGEMENT-1]) used by an authenticator to generate authentication assertions. A public key credential source consists of a struct with the following items:

type
whose value is of PublicKeyCredentialType, defaulting to public-key.

id
A Credential ID.

privateKey
The credential private key.

rpId
The Relying Party Identifier, for the Relying Party this public key credential source is associated with.

userHandle
The user handle associated when this public key credential source was created. This item is nullable.

otherUI
Optional other information used by the authenticator to inform its UI. For example, this might include the user's displayName.

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"
Credential 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 in 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. This default MAY be overridden by the caller, as long as the caller-specified RP ID value is a registrable domain suffix of or is equal 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:

  + The scheme is always https (i.e., a restriction), and,
  + the host may be equal to the Relying Party's origin's effective domain, or it may be equal to a registrable domain suffix of the Relying Party's origin's effective domain (i.e., an available relaxation), and,
  + all (TCP) ports on that host (i.e., a relaxation).

  This is done in order to match the behavior of pervasively deployed ambient credentials (e.g., cookies, [RFC6265]). Please note that this is a greater relaxation of "same-origin" restrictions than what document.domain's setter provides.

Test of User Presence
: A test of user presence is a simple form of authorization gesture and technical process where a user interacts with an authenticator by (typically) simply touching it (other modalities may also exist), yielding a boolean result. Note that this does not constitute user verification because a user

ID, is able to employ the public key credential in 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. This default MAY be overridden by the caller, as long as the caller-specified RP ID value is a registrable domain suffix of or is equal 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:

  + The scheme is always https (i.e., a restriction), and,
  + the host may be equal to the Relying Party's origin's effective domain, or it may be equal to a registrable domain suffix of the Relying Party's origin's effective domain (i.e., an available relaxation), and,
  + all (TCP) ports on that host (i.e., a relaxation).

  This is done in order to match the behavior of pervasively deployed ambient credentials (e.g., cookies, [RFC6265]). Please note that this is a greater relaxation of "same-origin" restrictions than what document.domain's setter provides.

Test of User Presence
: A test of user presence is a simple form of authorization gesture and technical process where a user interacts with an authenticator by (typically) simply touching it (other modalities may also exist), yielding a boolean result. Note that this does not constitute user verification because 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) [ISOBiometricVocabulary]. 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
UP
: Upon successful completion of a user presence test, the user is said to be "present".

User Verified
UV
: 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 user and are managed by an authenticator, with which the Relying Party interacts through the client (consisting of the browser and underlying OS platform). Scripts can (with the user's consent) request the browser to create a new credential for future use by the Relying Party. Scripts can also request the user's permission to perform authentication operations with an existing credential. All such operations are performed in the authenticator and are mediated by the

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) [ISOBiometricVocabulary]. 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
UP
: Upon successful completion of a user presence test, the user is said to be "present".

User Verified
UV
: 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. A WebAuthn Client is an intermediary entity typically implemented in the user agent (in whole, or in part). Conceptually, it underlies the Web Authentication API and embodies the implementation of the [[Create]](origin, options, sameOriginWithAncestors) and [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) internal methods. It is responsible for both marshalling the inputs for the underlying authenticator operations, and for returning the results of the latter operations to the Web Authentication API's callers.

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 user and are managed by an authenticator, with which the Relying Party interacts through the client (consisting of the browser and underlying OS platform). Scripts can (with the user's consent) request the browser to create a new credential for future use by the Relying Party. Scripts can also request the user's permission to perform authentication operations with an existing credential. All such operations are performed in the authenticator 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 credentials, ensures that all operations are scoped to a particular origin, and cannot be replayed against a different origin, by incorporating the origin in its responses. Specifically, as defined in 6.2 Authenticator operations, the full origin of the requester is included, and signed over, in the attestation object produced when a new credential is created as well as in all assertions produced by WebAuthn credentials.

Additionally, to maintain user privacy and prevent malicious Relying Parties from probing for the presence of 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 for each operation. Since this is an integral part of the WebAuthn security model, user agents only expose this API to callers in secure contexts.

The Web Authentication API is defined by the union of the Web IDL fragments presented in the following sections. A combined IDL listing is given in the IDL Index.

5.1. PublicKeyCredential Interface

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

```
[SecureContext, Exposed=Window]
interface PublicKeyCredential : Credential {
    [SameObject] readonly attribute ArrayBuffer              rawId;
    [SameObject] readonly attribute AuthenticatorResponse    response;
    AuthenticationExtensions getClientExtensionResults();
};
```

id
: This attribute is inherited from Credential, though PublicKeyCredential overrides Credential's getter, instead returning the base64url encoding of the data contained in the object's [[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

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 credentials, ensures that all operations are scoped to a particular origin, and cannot be replayed against a different origin, by incorporating the origin in its responses. Specifically, as defined in 6.2 Authenticator operations, the full origin of the requester is included, and signed over, in the attestation object produced when a new credential is created as well as in all assertions produced by WebAuthn credentials.

Additionally, to maintain user privacy and prevent malicious Relying Parties from probing for the presence of 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 for each operation. Since this is an integral part of the WebAuthn security model, user agents only expose this API to callers in secure contexts.

The Web Authentication API is defined by the union of the Web IDL fragments presented in the following sections. A combined IDL listing is given in the IDL Index.

5.1. PublicKeyCredential Interface

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

```
[SecureContext, Exposed=Window]
interface PublicKeyCredential : Credential {
    [SameObject] readonly attribute ArrayBuffer              rawId;
    [SameObject] readonly attribute AuthenticatorResponse    response;
    AuthenticationExtensionsClientOutputs getClientExtensionResults();
};
```

id
: This attribute is inherited from Credential, though PublicKeyCredential overrides Credential's getter, instead returning the base64url encoding of the data contained in the object's [[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 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 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 for use, and is therefore expected to be globally unique with high probability across all credentials of the same type, across all authenticators. This API does not constrain the format or length of this identifier, except that it must be sufficient for the platform to uniquely select a key. For example, an authenticator without on-board storage may create identifiers containing a credential 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 upon 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).

5.1.1. CredentialCreationOptions Extension

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

```
partial dictionary CredentialCreationOptions {
    MakePublicKeyCredentialOptions      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 {
    PublicKeyCredentialRequestOptions      publicKey;
};
```

5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method

generate an authentication assertion. If the PublicKeyCredential is created in response to create(), this attribute's value will 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 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 the credential ID, chosen by the platform with help from the authenticator. The credential ID is used to look up credentials for use, and is therefore expected to be globally unique with high probability across all credentials of the same type, across all authenticators.

  Note: This API does not constrain the format or length of this identifier, except that it MUST be sufficient for the platform to uniquely select a key. For example, an authenticator without on-board storage may create identifiers containing a credential 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 upon 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).

5.1.1. 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]](origin, options, sameOriginWithAncestors) method

PublicKeyCredential's interface object's implementation of the

[[Create]](origin, options, sameOriginWithAncestors) internal method [CREDENTIAL-MANAGEMENT-1] allows Relying Party scripts to call navigator.credentials.create() to request the creation of a new public key credential source, bound to an authenticator. This navigator.credentials.create() operation can be aborted by leveraging the AbortController; see DOM 3.3 Using AbortController and AbortSignal objects in APIs for detailed instructions.

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 MakePublicKeyCredentialOptions 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 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 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. Let 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 timer lifetimeTimer to this adjusted value. If the timeout member of options is not present, then set lifetimeTimer 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, ipv4 address, ipv6 address, opaque host, or empty host. Only the domain format of host is allowed here.
7. If options.rp.id

   Is present
   : If options.rp.id is not a registrable domain suffix of and is not equal to effectiveDomain, return a DOMException whose name is "SecurityError", and terminate this

PublicKeyCredential's interface object's implementation of the

[[Create]](origin, options, sameOriginWithAncestors) internal method [CREDENTIAL-MANAGEMENT-1] allows Relying Party scripts to call navigator.credentials.create() to request the creation of a new public key credential source, bound to an authenticator. This navigator.credentials.create() operation can be aborted by leveraging the AbortController; see DOM 3.3 Using AbortController and AbortSignal objects in APIs for detailed instructions.

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 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 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. Let 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 timer lifetimeTimer to this adjusted value. If the timeout member of options is not present, then set lifetimeTimer 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, ipv4 address, ipv6 address, opaque host, or empty host. Only the domain format of host is allowed here.
7. If options.rp.id

   Is present
   : If options.rp.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.

Is not present
Set options.rp.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().

8. Let credTypesAndPubKeyAlgs be a new list whose items are pairs of PublicKeyCredentialType and a COSEAlgorithmIdentifier.
9. For each current of options.pubKeyCredParams:
   1. If current.type does not contain a PublicKeyCredentialType supported by this implementation, then continue.
   2. Let alg be current.alg.
   3. Append the pair of current.type and alg to credTypesAndPubKeyAlgs.
10. If credTypesAndPubKeyAlgs is empty and options.pubKeyCredParams is not empty, return a DOMException whose name is "NotSupportedError", and terminate this algorithm.
11. Let clientExtensions be a new map and let authenticatorExtensions be a new map.
12. If the extensions member of options is present, then for each extensionId -> clientExtensionInput of options.extensions:
    1. If extensionId is not supported by this client platform or is not a registration extension, then continue.
    2. Set clientExtensions[extensionId] to clientExtensionInput.
    3. If extensionId is not an authenticator extension, then continue.
    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.
13. 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.

    tokenBindingId
    The Token Binding ID associated with callerOrigin, if one is available.

    clientExtensions
    clientExtensions

    authenticatorExtensions
    authenticatorExtensions

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.

algorithm.

Is not present
Set options.rp.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().

8. Let credTypesAndPubKeyAlgs be a new list whose items are pairs of PublicKeyCredentialType and a COSEAlgorithmIdentifier.
9. For each current of options.pubKeyCredParams:
   1. If current.type does not contain a PublicKeyCredentialType supported by this implementation, then continue.
   2. Let alg be current.alg.
   3. Append the pair of current.type and alg to credTypesAndPubKeyAlgs.
10. If credTypesAndPubKeyAlgs is empty and options.pubKeyCredParams is not empty, return a DOMException whose name is "NotSupportedError", and terminate this algorithm.
11. Let clientExtensions be a new map and let authenticatorExtensions be a new map.
12. If the extensions member of options is present, then for each extensionId -> clientExtensionInput of options.extensions:
    1. If extensionId is not supported by this client platform or is not a registration extension, then continue.
    2. Set clientExtensions[extensionId] to clientExtensionInput.
    3. If extensionId is not an authenticator extension, then continue.
    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.
13. 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.

    tokenBinding
    The status of Token Binding between the client and the callerOrigin, as well as the Token Binding ID associated with callerOrigin, if one is available.

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:
The definitions of "lifetime of" and "becomes available" are intended to represent how devices are hotplugged 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.authenticatorSelection is present:
        1. If options.authenticatorSelection.authenticatorAttachment is present and its value is not equal to authenticator's attachment modality, continue.
        2. If options.authenticatorSelection.requireResidentKey is set to true and the authenticator is not capable of storing a Client-Side-Resident Credential Private Key, continue.
        3. If options.authenticatorSelection.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 credential creation, a Boolean value, as follows. If options.authenticatorSelection.userVerification

        is set to required
            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 excludeCredentialDescriptorList be a new list.
    5. For each credential descriptor C in options.excludeCredentials:
        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.
    6. Invoke the authenticatorMakeCredential operation on authenticator with clientDataHash, options.rp, options.user, options.authenticatorSelection.requireResidentKey, userPresence, userVerification, credTypesAndPubKeyAlgs, excludeCredentialDescriptorList, and authenticatorExtensions as parameters.
    7. Append authenticator to issuedRequests.
20. 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 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

19. For each authenticator that becomes available on this platform during the lifetime of lifetimeTimer, do the following:
The definitions of "lifetime of" and "becomes available" are intended to represent how devices are hot-plugged into (USB) or discovered by (NFC) browsers, and are underspecified. Resolving this with good definitions or some other means will be addressed by resolving Issue #613.
    1. If options.authenticatorSelection is present:
        1. If options.authenticatorSelection.authenticatorAttachment is present and its value is not equal to authenticator's attachment modality, continue.
        2. If options.authenticatorSelection.requireResidentKey is set to true and the authenticator is not capable of storing a Client-Side-Resident Credential Private Key, continue.
        3. If options.authenticatorSelection.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 credential creation, a Boolean value, as follows. If options.authenticatorSelection.userVerification

        is set to required
            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 excludeCredentialDescriptorList be a new list.
    5. For each credential descriptor C in options.excludeCredentials:
        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.
    6. Invoke the authenticatorMakeCredential operation on authenticator with clientDataHash, options.rp, options.user, options.authenticatorSelection.requireResidentKey, userPresence, userVerification, credTypesAndPubKeyAlgs, excludeCredentialDescriptorList, and authenticatorExtensions as parameters.
    7. Append authenticator to issuedRequests.
20. While lifetimeTimer has not expired, 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 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.

If any authenticator returns an error status,

Remove authenticator from issuedRequests.

If any authenticator indicates success,

1. Remove authenticator from issuedRequests.
2. Let credentialCreationData be a struct whose items are:

    attestationObjectResult
        whose value is the bytes returned from the successful authenticatorMakeCredential operation.

        Note: this value is attObj, as defined in 6.3.4 Generating an Attestation Object.

    clientDataJSONResult
        whose value is the bytes of clientDataJSON.

    attestationConveyancePreferenceOption
        whose value is the value of options.attestation.

    clientExtensionResults
        whose value is an AuthenticationExtensions object containing extension identifier -> client extension output entries. The entries are created by running each extension's client

        extension processing algorithm to create the client extension outputs, for each client extension in clientDataJSON.clientExtensions.

3. Let constructCredentialAlg be an algorithm that takes a global object global, and whose steps are:

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 manifests this state to users is unspecified.

If any authenticator returns an error status equivalent to "InvalidStateError",

1. Remove authenticator from issuedRequests.
2. For each remaining authenticator in issuedRequests invoke the authenticatorCancel operation on authenticator and remove it from issuedRequests.
3. Return a DOMException whose name is "InvalidStateError" and terminate this algorithm.

Note: This error status is handled separately because the authenticator returns it only if excludeCredentialDescriptorList identifies a credential bound to the authenticator and the user has consented to the operation. Given this explicit consent, it is acceptable for this case to be distinguishable to the Relying Party.

If any authenticator returns an error status not equivalent to "InvalidStateError",

Remove authenticator from issuedRequests.

Note: This case does not imply user consent for the operation, so details about the error must be hidden from the Relying Party in order to prevent leak of potentially identifying information. See 14.2 Registration Ceremony Privacy for details.

If any authenticator indicates success,

1. Remove authenticator from issuedRequests.
2. Let credentialCreationData be a struct whose items are:

    attestationObjectResult
        whose value is the bytes returned from the successful authenticatorMakeCredential operation.

        Note: this value is attObj, as defined in 6.3.4 Generating an Attestation Object.

    clientDataJSONResult
        whose value is the bytes of clientDataJSON.

    attestationConveyancePreferenceOption
        whose value is the value of options.attestation.

    clientExtensionResults
        whose value is an AuthenticationExtensionsClientOutputs object containing extension identifier -> client extension output entries. The entries are created by running each extension's client extension processing algorithm to create the client extension outputs, for each client extension in clientDataJSON.clientExtensions.

3. Let constructCredentialAlg be an algorithm that takes a global object global, and whose steps are:

/Users/jehodges/Documents/work/standards/W3C/webauthn/index-master-tr-5e63e57-WD-07.txt, Top line: 1134

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

"none"
Replace potentially uniquely identifying information (such as AAGUID and attestation certificates) in the attested credential data and attestation statement, respectively, with blinded versions of the same data.

need to define "blinding". See also #462. <https://github.com/w3c/webauthn/issues/694>

"indirect"
The client MAY replace the AAGUID and attestation 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 attestation statement, unaltered, to the RP.

@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".

3. Let id be attestationObject.authData.attestedCredentialData.credentialId.
4. Let pubKeyCred be a new PublicKeyCredential object associated with global whose fields are:

[[identifier]]
id

/Users/jehodges/Documents/work/standards/W3C/webauthn/index-master-tr-716a169-WD-08a.txt, Top line: 1243

1. If credentialCreationData.attestationConveyancePreferenceOption's value is

"none"
Replace potentially uniquely identifying information with non-identifying versions of the same:

1. If the AAGUID in the attested credential data is 16 zero bytes, credentialCreationData.attestationObjectResult.fmt is "packed", and "x5c" & "ecdaaKeyId" are both absent from credentialCreationData.attestationObjectResult, 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.attestationObjectResult.attStmt to be an empty CBOR map. (See 8.7 None Attestation Statement Format and 6.3.4 Generating an Attestation Object).

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

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

@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 an "AttestationNotPrivateError".

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

attestationObject
attestationObject

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

5. Return pubKeyCred.

4. 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 6.2.2 The authenticatorGetAssertion operation to sign a Relying 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]](origin, options, sameOriginWithAncestors) internal method inherits the default behavior of Credential.[[CollectFromCredentialStore]](), of returning an empty set.

5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin,

response
A new AuthenticatorAttestationResponse object associated with global whose fields are:

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

attestationObject
attestationObject

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

5. Return pubKeyCred.

4. 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.

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 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]](origin, options, sameOriginWithAncestors) internal method inherits the default behavior of Credential.[[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 options.publicKey member contains a 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.
2. 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 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. Let 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 timer lifetimeTimer to this adjusted value. If the timeout member of options is not present, then set lifetimeTimer 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, ipv4 address, ipv6 address, opaque host, or empty host. Only the domain format of host is allowed here.
7. If options.rpId is not present, then set rpId to effectiveDomain.
   Otherwise:
   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.
      Note: rpId 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.rpId when calling get().
8. Let clientExtensions be a new map and let authenticatorExtensions be a new map.
9. If the extensions member of options is present, then for each extensionId -> clientExtensionInput of options.extensions:

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 options.publicKey member contains a 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.
2. 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 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. Let 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 timer lifetimeTimer to this adjusted value. If the timeout member of options is not present, then set lifetimeTimer 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, ipv4 address, ipv6 address, opaque host, or empty host. Only the domain format of host is allowed here.
7. If options.rpId is not present, then set rpId to effectiveDomain.
   Otherwise:
   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.
      Note: rpId 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.rpId when calling get().
8. Let clientExtensions be a new map and let authenticatorExtensions be a new 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 authenticator extension, then continue.
4. Let authenticatorExtensionInput be the (CBOR) result of running extensionId's client extension processing algorithm on clientExtensionInput. If the algorithm returned an error, continue.
5. Set authenticatorExtensions[extensionId] to the base64url encoding of authenticatorExtensionInput.

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
clientExtensions

authenticatorExtensions
authenticatorExtensions

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 aborted 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.
16. Start lifetimeTimer.
17. For each authenticator that becomes available on this platform during the lifetime of lifetimeTimer, perform the following steps: The definitions of "lifetime of" and "becomes available" are intended to represent how devices are hotplugged 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 required
        Let userVerification be true.

        is set to preferred
        If the authenticator

            is capable of user verification

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 authenticator extension, then continue.
4. Let authenticatorExtensionInput be the (CBOR) result of running extensionId's client extension processing algorithm on clientExtensionInput. If the algorithm returned an error, continue.
5. Set authenticatorExtensions[extensionId] to the base64url encoding of authenticatorExtensionInput.

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.

tokenBinding
The status of Token Binding between the client and the callerOrigin, as well as the Token Binding ID associated with callerOrigin, if one is available.

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 aborted 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.
16. Start lifetimeTimer.
17. For each authenticator that becomes available on this platform during the lifetime of lifetimeTimer, perform the following steps: The definitions of "lifetime of" and "becomes available" are intended to represent how devices are hot-plugged into (USB) or discovered by (NFC) browsers, and are underspecified. 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 required
        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
                   here in 6.2.2 The authenticatorGetAssertion
                   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.
```

```
                    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. If options.allowCredentials

            is not empty

                1. Let allowCredentialDescriptorList be a new list.
                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.
                3. If allowCredentialDescriptorList is empty, continue.
                4. Let distinctTransports be a new ordered set.
                5. If allowCredentialDescriptorList has exactly one
                   value, let savedCredentialId be a new
                   PublicKeyCredentialDescriptor.id and set its value
                   to allowCredentialDescriptorList[0].id's value (see
                   here in 6.2.3 The authenticatorGetAssertion
                   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.

is empty
Using local configuration knowledge of the appropriate transport to use with authenticator, invoke the authenticatorGetAssertion operation on authenticator with rpId, clientDataHash, 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 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.

If any authenticator returns an error status,
Remove authenticator from issuedRequests.

If any authenticator indicates success,

1. Remove authenticator from issuedRequests.
2. Let assertionCreationData be a struct whose items 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 6.2.2 The authenticatorGetAssertion operation.

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.

is empty
Using local configuration knowledge of the appropriate transport to use with authenticator, invoke the authenticatorGetAssertion operation on authenticator with rpId, clientDataHash, 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.

5. Append authenticator to issuedRequests.

18. While lifetimeTimer has not expired, 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 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.
Note: Authenticators may return an indication of "the user cancelled the entire operation". How a user agent manifests this state to users is unspecified.

If any authenticator returns an error status,
Remove authenticator from issuedRequests.

If any authenticator indicates success,

1. Remove authenticator from issuedRequests.
2. Let assertionCreationData be a struct whose items 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 The authenticatorGetAssertion operation.

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 object containing extension identifier -> client extension output entries. The entries are created by running each extension's client extension processing algorithm to create the client extension outputs, for each client extension in clientDataJSON.clientExtensions.

3. Let constructAssertionAlg be an algorithm that takes a global object global, and whose steps are:
1. Let pubKeyCred be a new PublicKeyCredential object associated with global 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.clientDataJSONResult.

authenticatorData
A new ArrayBuffer, created using global's %ArrayBuffer%, containing the bytes of assertionCreationData.authenticatorDataResult.

signature
A new ArrayBuffer, created using global's %ArrayBuffer%, containing the bytes of assertionCreationData.signatureResult.

userHandle
A new ArrayBuffer, created using

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
If the authenticator returned a user handle, set the value of userHandleResult to be the bytes of the returned user handle. Otherwise, set the value of userHandleResult to null.

clientExtensionResults
whose value is an AuthenticationExtensionsClientOutputs object containing extension identifier -> client extension output entries. The entries are created by running each extension's client extension processing algorithm to create the client extension outputs, for each client extension in clientDataJSON.clientExtensions.

3. Let constructAssertionAlg be an algorithm that takes a global object global, and whose steps are:
1. Let pubKeyCred be a new PublicKeyCredential object associated with global 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.clientDataJSONResult.

authenticatorData
A new ArrayBuffer, created using global's %ArrayBuffer%, containing the bytes of assertionCreationData.authenticatorDataResult.

signature
A new ArrayBuffer, created using global's %ArrayBuffer%, containing the bytes of assertionCreationData.signatureResult.

userHandle
If assertionCreationData.userHandleResult is null, set this field to

global's %ArrayBuffer%, containing the bytes of assertionCreationData.userHandleResult.

[[clientExtensionsResults]]
A new ArrayBuffer, created using global's %ArrayBuffer%, containing the bytes of assertionCreationData.clientExtensionResults.

2. Return pubKeyCred.

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

5. Return constructAssertionAlg and terminate this algorithm.

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

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

5.1.5. Store an existing credential - PublicKeyCredential's [[Store]](credential, sameOriginWithAncestors) method

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.

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.

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

null. Otherwise, set this field to a new ArrayBuffer, created using global's %ArrayBuffer%, containing the bytes of assertionCreationData.userHandleResult.

[[clientExtensionsResults]]
A new ArrayBuffer, created using global's %ArrayBuffer%, containing the bytes of assertionCreationData.clientExtensionResults.

2. Return pubKeyCred.

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

5. Return constructAssertionAlg and terminate this algorithm.

19. 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.

5.1.5. Store an existing credential - PublicKeyCredential's [[Store]](credential, sameOriginWithAncestors) method

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.

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.

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. Preventing silent access to an existing credential - PublicKeyCredential's [[preventSilentAccess]](credential, sameOriginWithAncestors) method

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

This internal method accepts no arguments.

5.1.7. 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 configured the client to not create such credentials.
* 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 prompt.
* 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 to not 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.

```
partial interface PublicKeyCredential {
    static Promise < boolean > isUserVerifyingPlatformAuthenticatorAvailable();
};
```

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().

5.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

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

```
[SecureContext, 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

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 configured the client to not create such credentials.
* 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 prompt.
* 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 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 to not 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.

```
partial interface PublicKeyCredential {
    static Promise < boolean > isUserVerifyingPlatformAuthenticatorAvailable();
};
```

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().

5.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

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

```
[SecureContext, 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 both authenticator data and an attestation statement. The former contains the AAGUID, a unique credential ID, and the credential public key. The contents of the attestation statement are determined by the attestation statement format used by the authenticator. It also contains any additional information that the Relying Party's server requires to validate the attestation statement, as well as to decode and validate the authenticator data along with the JSON-serialized client data. For more details, see 6.3 Attestation, 6.3.4 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      signature;
    [SameObject] readonly attribute ArrayBuffer      userHandle;
};
```

clientDataJSON
This attribute, inherited from AuthenticatorResponse, contains the JSON-serialized client data (see 5.8.1 Client data used in WebAuthn signatures (dictionary CollectedClientData)) passed to the authenticator by the client in order to generate this assertion. The exact JSON serialization must be preserved, as the hash of the serialized client data has been computed over it.

authenticatorData, of type ArrayBuffer, readonly
This attribute contains the authenticator data returned by the authenticator. See 6.1 Authenticator data.

signature, of type ArrayBuffer, readonly
This attribute contains the raw signature returned from the authenticator. See 6.2.2 The authenticatorGetAssertion operation.

userHandle, of type ArrayBuffer, readonly
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;
};
```

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

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

credential. The exact JSON serialization must be preserved, as the hash of the serialized client data has been computed over it.

attestationObject, of type ArrayBuffer, readonly
This attribute contains an attestation object, which is opaque to, and cryptographically protected against tampering by, the client. The attestation object contains both authenticator data and an attestation statement. The former contains the AAGUID, a unique credential ID, and the credential public key. The contents of the attestation statement are determined by the attestation statement format used by the authenticator. It also contains any additional information that the Relying Party's server requires to validate the attestation statement, as well as to decode and validate the authenticator data along with the JSON-serialized client data. For more details, see 6.3 Attestation, 6.3.4 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      signature;
    [SameObject] readonly attribute ArrayBuffer?     userHandle;
};
```

clientDataJSON
This attribute, inherited from AuthenticatorResponse, contains the JSON-serialized client data (see 5.10.1 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 operation.

userHandle, of type ArrayBuffer, readonly, nullable
This attribute contains the user handle returned from the authenticator, or null if the authenticator did not return a user handle. See 6.2.3 The authenticatorGetAssertion operation.

5.3. Parameters for Credential Generation (dictionary PublicKeyCredentialParameters)

```
dictionary PublicKeyCredentialParameters {
    required PublicKeyCredentialType      type;
    required COSEAlgorithmIdentifier      alg;
};
```

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;

    unsigned long                                timeout;
    sequence<PublicKeyCredentialDescriptor>      excludeCredentials = [];
    AuthenticatorSelectionCriteria               authenticatorSelection;
    AttestationConveyancePreference              attestation = "none";
    AuthenticationExtensions                     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@example.com" or "+14255551234").

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

  Its value's id member contains the user handle for the account, specified by the Relying Party.

challenge, of type BufferSource
: This member contains a challenge intended to be used for generating the newly created credential's attestation object.

pubKeyCredParams, of type sequence<PublicKeyCredentialParameters>
: This member contains information about the desired properties of the credential to be created. The sequence is ordered from most preferred to least preferred. The platform makes a best-effort to create the most preferred credential that it can.

timeout, of type unsigned long
: 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.

excludeCredentials, of type sequence<PublicKeyCredentialDescriptor>, defaulting to None
:

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 PublicKeyCredentialCreationOptions)

```
dictionary PublicKeyCredentialCreationOptions {
    required PublicKeyCredentialRpEntity         rp;
    required PublicKeyCredentialUserEntity       user;

    required BufferSource                        challenge;
    required sequence<PublicKeyCredentialParameters>  pubKeyCredParams;

    unsigned long                                timeout;
    sequence<PublicKeyCredentialDescriptor>      excludeCredentials = [];
    AuthenticatorSelectionCriteria               authenticatorSelection;
    AttestationConveyancePreference              attestation = "none";
    AuthenticationExtensionsClientInputs         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.

challenge, of type BufferSource
: This member contains a challenge intended to be used for generating the newly created credential's attestation object.

pubKeyCredParams, of type sequence<PublicKeyCredentialParameters>
: This member contains information about the desired properties of the credential to be created. The sequence is ordered from most preferred to least preferred. The platform makes a best-effort to create the most preferred credential that it can.

timeout, of type unsigned long
: 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.

excludeCredentials, of type sequence<PublicKeyCredentialDescriptor>, defaulting to None
:

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.

authenticatorSelection, of type AuthenticatorSelectionCriteria
This member is intended for use by Relying Parties that wish to select the appropriate authenticators to participate in the create() operation.

attestation, of type AttestationConveyancePreference, defaulting to "none"
This member is intended for use by Relying Parties that wish to express their preference for attestation conveyance. The default is none.

extensions, of type AuthenticationExtensions
This member contains additional parameters requesting additional processing by the client and authenticator. For example, the caller may request that only authenticators with certain capabilies 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 "WebAuthn Extension Identifier" registry established by [WebAuthn-Registries] for an up-to-date list of registered WebAuthn Extensions.

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.

```
dictionary PublicKeyCredentialEntity {
    required DOMString    name;
    USVString             icon;
};
```

name, of type DOMString
A human-friendly 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 members'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 user's avatar or a Relying Party's logo. This URL MUST be an a priori authenticated URL. Authenticators MUST accept and store a 128 byte minimum length for a icon members's value. Authenticators MAY ignore a icon members's value if its length is greater than 128 byes.

5.4.2. RP Parameters for Credential Generation (dictionary

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.

authenticatorSelection, of type AuthenticatorSelectionCriteria
This member is intended for use by Relying Parties that wish to select the appropriate authenticators to participate in the create() operation.

attestation, of type AttestationConveyancePreference, defaulting to "none"
This member is intended for use by Relying Parties that wish to express their preference for attestation conveyance. The default is none.

extensions, of type AuthenticationExtensionsClientInputs
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 "WebAuthn Extension Identifier" registry established by [WebAuthn-Registries] for an up-to-date list of registered WebAuthn Extensions.

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.

```
dictionary PublicKeyCredentialEntity {
    required DOMString    name;
    USVString             icon;
};
```

name, of type DOMString
A human-readable name for the entity. Its function depends on what the PublicKeyCredentialEntity represents:

+ 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-palatable 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, "alexm", "alex.p.mueller@example.com" or "+14255551234". The Relying Party MAY let the user choose this, and MAY restrict the choice as needed or appropriate. For example, a Relying Party might choose to map human-palatable username account identifiers to the name member of PublicKeyCredentialUserEntity.

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 user's avatar or a Relying Party's logo. This URL MUST be an a priori authenticated URL. 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

```
1901      PublicKeyCredentialRpEntity)
1902
1903    The PublicKeyCredentialRpEntity dictionary is used to supply additional
1904    Relying Party attributes when creating a new credential.
1905  dictionary PublicKeyCredentialRpEntity : PublicKeyCredentialEntity {
1906    DOMString     id;
1907  };
1908
1909    id, of type DOMString
1910        A unique identifier for the Relying Party entity, which sets the
1911        RP ID.
1912
1913     5.4.3. User Account Parameters for Credential Generation (dictionary
1914     PublicKeyCredentialUserEntity)
1915
1916    The PublicKeyCredentialUserEntity dictionary is used to supply
1917    additional user account attributes when creating a new credential.
1918  dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
1919    required BufferSource   id;
1920    required DOMString      displayName;
1921  };
1922
1923    id, of type BufferSource
1924        The user handle of the user account entity.
1925
1926    displayName, of type DOMString
1927        A friendly name for the user account (e.g., "John P. Smith").
1928        Authenticators MUST accept and store a 64 byte minimum length
1929        for a displayName members's value. Authenticators MAY truncate a
1930        displayName member's value to a length equal to or greater than
1931        64 bytes.
1932
1933     5.4.4. Authenticator Selection Criteria (dictionary
1934     AuthenticatorSelectionCriteria)
1935
1936    Relying Parties may use the AuthenticatorSelectionCriteria dictionary
1937    to specify their requirements regarding authenticator attributes.
1938  dictionary AuthenticatorSelectionCriteria {
1939    AuthenticatorAttachment     authenticatorAttachment;
1940    boolean                     requireResidentKey = false;
1941    UserVerificationRequirement  userVerification = "preferred";
1942  };
1943
1944    authenticatorAttachment, of type AuthenticatorAttachment
1945        If this member is present, eligible authenticators are filtered
1946        to only authenticators attached with the specified 5.4.5
1947        Authenticator Attachment enumeration (enum
1948        AuthenticatorAttachment).
1949
1950    requireResidentKey, of type boolean, defaulting to false
1951        This member describes the Relying Parties' requirements
1952        regarding availability of the Client-side-resident Credential
1953        Private Key. If the parameter is set to true, the authenticator
1954        MUST create a Client-side-resident Credential Private Key when
1955        creating a public key credential.
1956
1957    userVerification, of type UserVerificationRequirement, defaulting to
1958        "preferred"
1959        This member describes the Relying Party's requirements regarding
1960        user verification for the create() operation. Eligible
1961        authenticators are filtered to only those capable of satisfying
1962        this requirement.
1963
1964     5.4.5. Authenticator Attachment enumeration (enum AuthenticatorAttachment)
1965
1966  enum AuthenticatorAttachment {
```

```
2049      PublicKeyCredentialRpEntity)
2050
2051    The PublicKeyCredentialRpEntity dictionary is used to supply additional
2052    Relying Party attributes when creating a new credential.
2053  dictionary PublicKeyCredentialRpEntity : PublicKeyCredentialEntity {
2054    DOMString     id;
2055  };
2056
2057    id, of type DOMString
2058        A unique identifier for the Relying Party entity, which sets the
2059        RP ID.
2060
2061     5.4.3. User Account Parameters for Credential Generation (dictionary
2062     PublicKeyCredentialUserEntity)
2063
2064    The PublicKeyCredentialUserEntity dictionary is used to supply
2065    additional user account attributes when creating a new credential.
2066  dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
2067    required BufferSource   id;
2068    required DOMString      displayName;
2069  };
2070
2071    id, of type BufferSource
2072        The user handle of the user account entity.
2073
2074    displayName, of type DOMString
2075        A human-friendly name for the user account, intended only for
2076        display. For example, "Alex P. Mller" or " ". The Relying
2077        Party SHOULD let the user choose this, and SHOULD NOT restrict
2078        the choice more than necessary.
2079
2080        Authenticators MUST accept and store a 64-byte minimum length
2081        for a displayName member's value. Authenticators MAY truncate a
2082        displayName member's value to a length equal to or greater than
2083        64 bytes.
2084
2085     5.4.4. Authenticator Selection Criteria (dictionary
2086     AuthenticatorSelectionCriteria)
2087
2088    Relying Parties may use the AuthenticatorSelectionCriteria dictionary
2089    to specify their requirements regarding authenticator attributes.
2090  dictionary AuthenticatorSelectionCriteria {
2091    AuthenticatorAttachment     authenticatorAttachment;
2092    boolean                     requireResidentKey = false;
2093    UserVerificationRequirement  userVerification = "preferred";
2094  };
2095
2096    authenticatorAttachment, of type AuthenticatorAttachment
2097        If this member is present, eligible authenticators are filtered
2098        to only authenticators attached with the specified 5.4.5
2099        Authenticator Attachment enumeration (enum
2100        AuthenticatorAttachment).
2101
2102    requireResidentKey, of type boolean, defaulting to false
2103        This member describes the Relying Parties' requirements
2104        regarding availability of the Client-side-resident Credential
2105        Private Key. If the parameter is set to true, the authenticator
2106        MUST create a Client-side-resident Credential Private Key when
2107        creating a public key credential.
2108
2109    userVerification, of type UserVerificationRequirement, defaulting to
2110        "preferred"
2111        This member describes the Relying Party's requirements regarding
2112        user verification for the create() operation. Eligible
2113        authenticators are filtered to only those capable of satisfying
2114        this requirement.
2115
2116     5.4.5. Authenticator Attachment enumeration (enum AuthenticatorAttachment)
2117
2118  enum AuthenticatorAttachment {
```

```
    "platform",      // Platform attachment
    "cross-platform"  // Cross-platform attachment
};
```

Clients may communicate with authenticators using a variety of mechanisms. For example, a client may use a platform-specific API to communicate with an authenticator which is physically bound to a platform. On the other hand, a client may use a variety of standardized cross-platform transport protocols such as Bluetooth (see 5.8.4 Authenticator Transport enumeration (enum AuthenticatorTransport)) 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.

* cross-platform attachment - the respective authenticator is attached using cross-platform transports. Authenticators of this class are removable from, and can "roam" among, client platforms.

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

5.4.6. Attestation Conveyance Preference enumeration (enum AttestationConveyancePreference)

Relying Parties may use AttestationConveyancePreference to specify their preference regarding attestation conveyance during credential generation.

```
enum AttestationConveyancePreference {
    "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 uniquely identifying information to the Relying Party, or to save a roundtrip to a Privacy CA.
  This is the default value.


```
    "platform",      // Platform attachment
    "cross-platform"  // Cross-platform attachment
};
```

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 can use a variety of standardized cross-platform transport protocols such as Bluetooth (see 5.10.4 Authenticator Transport enumeration (enum AuthenticatorTransport)) 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 platform credential may be used by Relying Parties to quickly and conveniently reauthenticate the user with a minimum of friction, e.g., the user will not have to dig around in their pocket for their key fob or phone. As a concrete example of the latter, when the user is accessing the Relying Party from a given client for the first time, they may be asked to use a roaming credential which was originally registered with the Relying Party using a different client.

Note: An attachment modality selection option is available only in the [[Create]](origin, options, sameOriginWithAncestors) operation. The Relying Party may use it to, for example, ensure the user has a roaming credential for authenticating using other clients; or to specifically register a platform credential for easier reauthentication using 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 AttestationConveyancePreference)

Relying Parties may use AttestationConveyancePreference to specify their preference regarding attestation conveyance during credential generation.

```
enum AttestationConveyancePreference {
    "none",
    "indirect",
    "direct"
};
```

* none - indicates that the Relying Party is not interested in authenticator attestation. For example, in order to potentially 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 a Privacy CA,
        in order to protect the user's privacy, or to assist Relying
        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
  PublicKeyCredentialRequestOptions)

  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;
    unsigned long                      timeout;
    USVString                          rpId;
    sequence<PublicKeyCredentialDescriptor> allowCredentials = [];
    UserVerificationRequirement        userVerification = "preferred";
    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 decending order of the
      caller's preference (the first item in the list is the most
      preferred credential, and so on down the list).

  userVerification, of type UserVerificationRequirement, defaulting to
      "preferred"
      This member describes the Relying Party's requirements regarding
      user verification for the get() operation. Eligible
      authenticators are filtered to only those capable of satisfying
      this requirement.

  extensions, of type AuthenticationExtensions
      This optional member contains additional parameters requesting
      additional processing by the client and authenticator. For
      example, if transaction confirmation is sought from the user,
      then the prompt string might be included as an extension.

  5.6. Abort operations with AbortSignal

  Developers are encouraged to leverage the AbortController to manage the
```

```
      * 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 Relying 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
  PublicKeyCredentialRequestOptions)

  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;
    unsigned long                      timeout;
    USVString                          rpId;
    sequence<PublicKeyCredentialDescriptor> allowCredentials = [];
    UserVerificationRequirement        userVerification = "preferred";
    AuthenticationExtensionsClientInputs 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 the
      caller's preference (the first item in the list is the most
      preferred credential, and so on down the list).

  userVerification, of type UserVerificationRequirement, defaulting to
      "preferred"
      This member describes the Relying Party's requirements regarding
      user verification for the get() operation. Eligible
      authenticators are filtered to only those capable of satisfying
      this requirement.

  extensions, of type AuthenticationExtensionsClientInputs
      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.

  5.6. Abort operations with AbortSignal

  Developers are encouraged to leverage the AbortController to manage 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 AbortController must reject the promise immediately once the aborted flag is set. Given the complex inheritance and parallelization structure of the [[Create]](origin, options, sameOriginWithAncestors) and [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) methods, the algorithms for the two APIs fulfills this requirement by checking the aborted flag in three places. In the case of [[Create]](origin, options, sameOriginWithAncestors), the aborted flag is checked first in 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. The same goes for [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors).

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) 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 focuses. If a hook is provided, the above paragraph will be updated to include the hook. See WHATWG HTML WG Issue #2711 for more details.

5.7. Authentication Extensions (typedef AuthenticationExtensions)

```
typedef record<DOMString, any>      AuthenticationExtensions;
```

This is a dictionary containing zero or more WebAuthn extensions, as defined in 9 WebAuthn Extensions. An AuthenticationExtensions instance can contain either client extensions or authenticator extensions, depending upon context.

5.8. Supporting Data Structures

The public key credential type uses certain data structures that are

[[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 AbortController must reject the promise immediately once the aborted flag is set. Given the complex inheritance and parallelization structure of the [[Create]](origin, options, sameOriginWithAncestors) and [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) methods, the algorithms for the two APIs fulfills this requirement by checking the aborted flag in three places. In the case of [[Create]](origin, options, sameOriginWithAncestors), the aborted flag is checked first in 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. The same goes for [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors).

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) 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 focuses. If a hook is provided, the above paragraph will be updated to include the hook. See WHATWG HTML WG Issue #2711 for more details.

5.7. Authentication Extensions Client Inputs (typedef AuthenticationExtensionsClientInputs)

```
dictionary AuthenticationExtensionsClientInputs {
};
```

This is a dictionary containing the client extension input values for zero or more WebAuthn extensions, as defined in 9 WebAuthn Extensions.

5.8. Authentication Extensions Client Outputs (typedef AuthenticationExtensionsClientOutputs)

```
dictionary AuthenticationExtensionsClientOutputs {
};
```

This is a dictionary containing the client extension output values for zero or more WebAuthn extensions, as defined in 9 WebAuthn Extensions.

5.9. Authentication Extensions Authenticator Inputs (typedef AuthenticationExtensionsAuthenticatorInputs)

```
typedef record<DOMString, DOMString> AuthenticationExtensionsAuthenticatorInputs
;
```

This is a dictionary containing the authenticator extension input values for zero or more WebAuthn extensions, as defined in 9 WebAuthn Extensions.

5.10. Supporting Data Structures

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.

```
dictionary CollectedClientData {
    required DOMString          type;
    required DOMString          challenge;
    required DOMString          origin;
    required DOMString          hashAlgorithm;
    DOMString                   tokenBindingId;
    AuthenticationExtensions    clientExtensions;
    AuthenticationExtensions    authenticatorExtensions;
};
```

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 Relying 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 hashAlgorithm) of the JSON-serialized client data, as constructed by the client.

5.8.2. Credential Type enumeration (enum PublicKeyCredentialType)

```
enum PublicKeyCredentialType {
    "public-key"
```


specified in supporting specifications. These are as follows.

5.10.1. 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 can be any type that has a valid encoding in JSON. Its structure is defined by the following Web IDL.

```
dictionary CollectedClientData {
    required DOMString          type;
    required DOMString          challenge;
    required DOMString          origin;
    TokenBinding                tokenBinding;
};

dictionary TokenBinding {
    required TokenBindingStatus status;
    DOMString id;
};

enum TokenBindingStatus { "present", "supported", "not-supported" };
```

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 tokenBinding member contains information about the state of the Token Binding protocol used when communicating with the Relying Party. The status member is one of:

* not-supported: when the client does not support token binding.
* supported: the client supports token binding, but it was not negotiated when communicating with the Relying Party.
* present: token binding was used when communicating with the Relying Party. In this case, the id member MUST be present and MUST be a base64url encoding of the Token Binding ID that was used.

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.

5.10.2. Credential Type enumeration (enum PublicKeyCredentialType)

```
enum PublicKeyCredentialType {
    "public-key"
```

```
};

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

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

   5.8.3. Credential Descriptor (dictionary PublicKeyCredentialDescriptor)

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.

   5.8.4. Authenticator Transport enumeration (enum AuthenticatorTransport)

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

  Authenticators may communicate with Clients using a variety of
  transports. This enumeration defines a hint as to how Clients might
  communicate with a particular Authenticator in order to obtain an
  assertion for a specific credential. Note that these hints represent
  the Relying Party's best belief as to how an Authenticator may be
  reached. A Relying Party may obtain a list of transports hints from
  some attestation statement formats or via some out-of-band mechanism;
  it is outside the scope of this specification to define that mechanism.
    * usb - the respective Authenticator may be contacted over USB.
    * nfc - the respective Authenticator may be contacted over Near Field
      Communication (NFC).
    * ble - the respective Authenticator may be contacted over Bluetooth
      Smart (Bluetooth Low Energy / BLE).

   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
```



```
};

  This enumeration defines the valid credential types. It is an extension
  point; values can 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)

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 public key 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 public key credential the
  caller is referring to.

  The id member contains the credential ID of the public key credential
  that the caller is referring to.

   5.10.4. Authenticator Transport enumeration (enum AuthenticatorTransport)

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

  Authenticators may communicate with clients using a variety of
  transports. This enumeration defines a hint as to how clients might
  communicate with a particular authenticator in order to obtain an
  assertion for a specific credential. Note that these hints represent
  the Relying Party's best belief as to how an authenticator may be
  reached. A Relying Party may obtain a list of transports hints from
  some attestation statement formats or via some out-of-band mechanism;
  it is outside the scope of this specification to define that mechanism.
    * usb - the respective authenticator can be contacted over USB.
    * nfc - the respective authenticator can be contacted over Near Field
      Communication (NFC).
    * ble - the respective authenticator can be contacted over Bluetooth
      Smart (Bluetooth Low Energy / BLE).

   5.10.5. 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".

   5.10.6. 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 not 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).

6. WebAuthn Authenticator model

The API defined in this specification implies a specific abstract functional model for an authenticator. This section describes the authenticator model.

Client platforms may implement and expose this abstract model in any way desired. However, the behavior of the client's Web Authentication API implementation, when operating on the authenticators supported by that platform, MUST be indistinguishable from the behavior specified in 5 Web Authentication API.

For authenticators, this model defines the logical operations that they must support, and the data formats that they expose to the client and the Relying Party. However, it does not define the details of how authenticators communicate with the client platform, unless they are required for interoperability with Relying Parties. For instance, 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 may itself 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 in those cases this cryptographic module (which may, for example, be a TPM) could be considered more trustworthy than the rest of the authenticator.

Each authenticator stores some number of 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 credential is also associated with a Relying Party, whose identity is represented by a Relying Party Identifier (RP ID).

Each authenticator has an AAGUID, which is a 128-bit identifier that indicates the type (e.g. make and model) of the authenticator. The AAGUID MUST be chosen by the manufacturer to be identical across all substantially identical authenticators made by that manufacturer, and different (with probability 1-2^-128 or greater) from the AAGUIDs of

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 not 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).

6. WebAuthn Authenticator Model

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

Client platforms MAY implement and expose this abstract model in any way desired. However, the behavior of the client's Web Authentication API implementation, when operating on the authenticators supported by that platform, MUST be indistinguishable from the behavior specified in 5 Web Authentication API.

For authenticators, this model defines the logical operations that they MUST support, and the data formats that they expose to the client and the Relying Party. However, it does not define the details of how authenticators communicate with the client platform, unless they are necessary 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.

Relying Parties may influence authenticator selection, if they deem necessary, by stipulating various authenticator characteristics when creating credentials and/or when generating assertions, through use of credential creation options or assertion generation options, respectively. The algorithms underlying the WebAuthn API marshal these options and pass them to the applicable authenticator operations defined below.

In this abstract model, the authenticator provides key management and cryptographic signatures. It can 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 in those cases this cryptographic module (which may, for example, be a TPM) could be considered more trustworthy than the rest of the authenticator.

Each authenticator stores a credentials map, a map from (rpId, [userHandle]) to public key credential source.

Additionally, each authenticator has an AAGUID, which is a 128-bit identifier indicating 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 (with probability 1-2^-128 or greater)

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 data are observed, and added at different levels of the stack as a signature request passes from the server to the authenticator. In verifying a signature, the server checks these bindings against expected values. These contextual bindings are divided in two: Those added by the RP or the client, referred to as client data; and those added by the authenticator, referred to as the authenticator data. The authenticator signs over the client data, but is otherwise not interested in its contents. To save bandwidth and processing requirements on the authenticator, the client hashes the client data and sends only the result to the authenticator. The authenticator signs over the combination of 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 authenticator type (as denoted by its AAGUID) and the credential public key. The attestation signature is signed by an attestation private key, which is chosen depending on the type of attestation desired. For more details on attestation, see 6.3 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. Thus, an assertion signature asserts that the authenticator possessing 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 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 security properties of the authenticator. In one extreme case, the authenticator may be embedded in the client, and its bindings may be no more trustworthy than the client data. At the other extreme, the authenticator may be a discrete entity with high-security hardware and

from the AAGUIDs of 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 data are observed and added at different levels of the stack as a signature request passes from the server to the authenticator. In verifying a signature, the server checks these bindings against expected values. These contextual bindings are divided in two: Those added by the RP or the client, referred to as client data; and those added by the authenticator, referred to as the authenticator data. The authenticator signs over the client data, but is otherwise not interested in its contents. To save bandwidth and processing requirements on the authenticator, the client hashes the client data and sends only the result to the authenticator. The authenticator signs over the combination of 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 authenticator and the credential. For instance, an attestation signature asserts the authenticator type (as denoted by its AAGUID) and the credential public key. The attestation signature is signed by an attestation private key, which is chosen depending on the type of attestation desired. For more details on attestation, see 6.3 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. Thus, an assertion signature asserts that the authenticator possessing 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 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 security properties of the authenticator. In one extreme case, the authenticator may be embedded in the client, and its bindings may be no more trustworthy than the client 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.

Name Length (in bytes) Description
rpIdHash 32 SHA-256 hash of the RP ID associated with the credential.
flags 1 Flags (bit 0 is the least significant bit):
* Bit 0: User Present (UP) result.
  + 1 means the user is present.
  + 0 means the user is not present.
* Bit 1: Reserved for future use (RFU1).
* Bit 2: User Verified (UV) result.
  + 1 means the user is verified.
  + 0 means the user is not verified.
* Bits 3-5: Reserved for future use (RFU2).
* Bit 6: Attested credential data included (AT).
  + Indicates whether the authenticator added attested credential data.
* Bit 7: Extension data included (ED).
  + Indicates if the authenticator data has extensions.

signCount 4 Signature counter, 32-bit unsigned big-endian integer.
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.
extensions variable (if present) Extension-defined authenticator data. This is a CBOR [RFC7049] map with extension identifiers as keys, and authenticator extension outputs as values. See 9 WebAuthn Extensions for details.

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 of 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 RFU bits 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 Authenticator data layout.

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.

Name Length (in bytes) Description
rpIdHash 32 SHA-256 hash of the RP ID associated with the credential.
flags 1 Flags (bit 0 is the least significant bit):
* Bit 0: User Present (UP) result.
  + 1 means the user is present.
  + 0 means the user is not present.
* Bit 1: Reserved for future use (RFU1).
* Bit 2: User Verified (UV) result.
  + 1 means the user is verified.
  + 0 means the user is not verified.
* Bits 3-5: Reserved for future use (RFU2).
* Bit 6: Attested credential data included (AT).
  + Indicates whether the authenticator added attested credential data.
* Bit 7: Extension data included (ED).
  + Indicates if the authenticator data has extensions.

signCount 4 Signature counter, 32-bit unsigned big-endian integer.
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.
extensions variable (if present) Extension-defined authenticator data. This is a CBOR [RFC7049] map with extension identifiers as keys, and authenticator extension outputs as values. See 9 WebAuthn Extensions for details.

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 of 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 RFU bits 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.

[fido-signature-formats-figure1.html] Authenticator data layout.

Note that the authenticator data describes its own length: If the AT and ED flags are not set, it is always 37 bytes long. The 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. Authenticators may implement a global signature counter, i.e., on a per-authenticator basis, but this is less privacy-friendly for users.
* should ensure that the signature counter value does not accidentally decrease (e.g., due to hardware failures).

6.2. Authenticator operations

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

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

6.2.1. The authenticatorMakeCredential operation

It takes the following input parameters:

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. Authenticators may implement a global signature counter, i.e., on a per-authenticator basis, but this is less privacy-friendly for users.
* 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 must maintain isolation between sessions. It may do this by only allowing one session to exist at any particular time, or by providing more complicated session management.

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

6.2.1. Lookup Credential Source by Credential ID algorithm

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 map:
   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. 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 supported. If not, return an error code equivalent to "NotSupportedError" and terminate the operation.
3. Check if any credential bound to this authenticator matches an item of excludeCredentialDescriptorList. A match occurs if a credential matches rpEntity.id and an excludeCredentialDescriptorList item's excludeCredentialDescriptorList.id and excludeCredentialDescriptorList.type. If so, return an error code equivalent to "NotAllowedError" and terminate the operation.

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 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 supported. If not, return an error code equivalent to "NotSupportedError" and terminate the operation.
3. 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

      confirms consent to create a new credential
      return an error code equivalent to "InvalidStateError" and terminate the operation.

      does not consent to create a new credential
      return an error code equivalent to "NotAllowedError"

4. If requireResidentKey is true and the authenticator cannot store a Client-side-resident Credential Private Key, 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.
   If requireUserVerification is true, the method of obtaining user consent MUST include user verification.
   If requireUserPresence is true, the method of obtaining user consent MUST include a test of user presence.
   If the user denies consent or if user verification fails, return an error code equivalent to "NotAllowedError" and terminate the operation.
7. 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 credTypesAndPubKeyAlgs 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 privateKey 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.

and terminate the operation.

4. If requireResidentKey is true and the authenticator cannot store a Client-side-resident Credential Private Key, 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.
   If requireUserVerification is true, the method of obtaining user consent MUST include user verification.
   If requireUserPresence is true, 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.
7. 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 credTypesAndPubKeyAlgs that is supported by this authenticator.
   2. Let userHandle be userEntity.id.
   3. Let credentialSource be a new public key credential source with the fields:

      type
          public-key.

      privateKey
          privateKey

      rpId
          rpEntity.id

      userHandle
          userHandle

      otherUI
          Any other information the authenticator chooses to include.

   4. If requireResidentKey is true or the authenticator chooses to create a Client-side-resident Credential Private Key:
      1. Let credentialId be a new credential id.
      2. Set credentialSource.id to credentialId.
      3. Let credentials be this authenticator's credentials map.
      4. Set credentials[(rpEntity.id, userHandle)] to credentialSource.
   5. Otherwise:
      1. Let credentialId be the result of serializing and encrypting credentialSource so that only this authenticator can decrypt it.
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 -> authenticator extension input 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 [[DiscoverFromExternalSource]](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.

Note: Before performing this operation, all other operations in progress in the authenticator session must be aborted by running the authenticatorCancel operation.

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

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.

6.2.3. 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 [[DiscoverFromExternalSource]](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 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 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.
2. Let credentialOptions be a new empty set of public key credential sources.
3. If allowCredentialDescriptorList was supplied, then for each descriptor of allowCredentialDescriptorList:
    1. Let credSource be the result of looking up descriptor.id in this authenticator.
    2. If credSource is not null, append it to credentialOptions.

match a credential bound to this authenticator. A match occurs if a credential matches rpId and an allowCredentialDescriptorList item's id and type members.

5. If allowCredentialDescriptorList is now empty, return an error code equivalent to "NotAllowedError" and terminate the operation.
6. Let selectedCredential be a credential as follows. If the size of allowCredentialDescriptorList

is exactly 1
Let selectedCredential be the credential matching allowCredentialDescriptorList[0].

is greater than 1
Prompt the user to select selectedCredential from the credentials matching the items in allowCredentialDescriptorList.

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.
If requireUserVerification is true, the method of obtaining user consent MUST include user verification.
If requireUserPresence is true, the method of obtaining user consent MUST include a test of user presence.
If the user denies consent or if user verification fails, return an error code equivalent to "NotAllowedError" and terminate the operation.
8. Let processedExtensions be the result of authenticator extension processing for each supported extension identifier/input pair in extensions.
9. Increment the RP ID-associated signature counter or the global signature counter value, depending on which approach is implemented by the authenticator, by some positive value.
10. Let authenticatorData be the byte array specified in 6.1 Authenticator data including processedExtensions, if any, as the extensions and excluding attestedCredentialData.
11. Let signature be the assertion signature of the concatenation authenticatorData || hash using the private key of 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.
Generating an assertion signature Generating an assertion signature.
12. If any error occurred while generating the assertion signature, return an error code equivalent to "UnknownError" and terminate the operation.
13. Return to the user agent:
+ selectedCredential's 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 values.
Note: If the client supplies a list of exactly one credential and it was successfully employed, then its credential ID is not returned since the client already knows it. This saves transmitting these bytes over what may be a constrained connection in what is likely a common case.
+ authenticatorData
+ signature
+ The user handle associated with selectedCredential.

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.

4. Otherwise (allowCredentialDescriptorList was not supplied), for each key -> credSource of this authenticator's credentials map, append credSource to credentialOptions.
5. Remove any items from credentialOptions whose rpId is not equal to rpId.
6. If credentialOptions is now empty, return an error code equivalent to "NotAllowedError" and terminate the operation.
7. Prompt the user to select a public key credential source selectedCredential from credentialOptions. 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.

If requireUserVerification is true, the method of obtaining user consent MUST include user verification.
If requireUserPresence is true, the method of obtaining user consent MUST include a test of user presence.
If the user does not consent, return an error code equivalent to "NotAllowedError" and terminate the operation.

8. Let processedExtensions be the result of authenticator extension processing for each supported extension identifier -> authenticator extension input in extensions.
9. Increment the RP ID-associated signature counter or the global signature counter value, depending on which approach is implemented by the authenticator, by some positive value.
10. Let authenticatorData be the byte array specified in 6.1 Authenticator data including processedExtensions, if any, as the extensions and excluding attestedCredentialData.
11. Let signature be the assertion signature of the concatenation authenticatorData || hash using the privateKey of 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.
[fido-signature-formats-figure2.html] Generating an assertion signature.
12. If any error occurred while generating the assertion signature, return an error code equivalent to "UnknownError" and terminate the operation.
13. Return to the user agent:
+ selectedCredential.id, if either a list of credentials (i.e., allowCredentialDescriptorList) of length 2 or greater was supplied by the client, or no such list was supplied.
Note: If, within allowCredentialDescriptorList, the client supplied exactly one credential and it was successfully employed, then its credential ID is not returned since the client already knows it. This saves transmitting these bytes over what may be a constrained connection in what is likely a common case.
+ 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.

6.2.3. 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 public key, an attestation statement verifable 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 illustrating the included authenticator data (containing attested credential data) and the attestation statement.

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 defines 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

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 defines 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:

* The attestation type, which determines the trust model,
* The attestation statement format, which may constrain the strength of the attestation by limiting what can be expressed in an attestation statement, and
* 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
aaguid	16	The AAGUID of the authenticator.
credentialIdLength	2	Byte length L of Credential ID
credentialId	L	Credential ID
credentialPublicKey	variable	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.

formats and types have more limited applicability.

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

* The attestation type, which determines the trust model,
* The attestation statement format, which MAY constrain the strength of the attestation by limiting what can be expressed in an attestation statement, and
* 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
aaguid	16	The AAGUID of the authenticator.
credentialIdLength	2	Byte length L of Credential ID, 16-bit unsigned big-endian integer.
credentialId	L	Credential ID
credentialPublicKey	variable	The credential public key encoded in COSE_Key format, as defined in Section 7 of [RFC8152], using the CTAP2 canonical CBOR encoding form. The COSE_Key-encoded credential public key MUST contain the optional "alg" parameter and MUST NOT contain any other optional parameters. The "alg" parameter MUST contain a COSEAlgorithmIdentifier value. The encoded credential public key MUST also contain any additional required parameters stipulated by the relevant key type specification, i.e., required for the key type "kty" and algorithm "alg" (see Section 8 of [RFC8152]).

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 for the credentialPublicKey value, and are presented in [CDDL] for clarity.

[RFC8152] Section 7 defines the general framework for all COSE_Key-encoded keys. Specific key types for specific algorithms are defined in other sections of [RFC8152] as well as in other specifications, as noted below.

Below is an example of a COSE_Key-encoded Elliptic Curve public key in EC2 format (see [RFC8152] Section 13.1), on the P-256 curve, to be used with the ES256 signature algorithm (ECDSA w/ SHA-256, see [RFC8152] Section 8.1):

```
{
  1:  2,  ; kty: EC2 key type
  3: -7,  ; alg: ES256 signature algorithm
 -1:  1,  ; crv: P-256 curve
 -2:  x,  ; x-coordinate as byte string 32 bytes in length
          ; e.g., in hex: 65eda5a12577c2bae829437fe338701a10aaa375e1bb5b5de108de439c08551d
 -3:  y   ; y-coordinate as byte string 32 bytes in length
```

; e.g., in hex: 1e52ed75701163f7f9e40ddf9f341b3dc9ba860af7e0ca7ca7e9e
ecd0084d19c
}

Below is the above Elliptic Curve public key encoded in the CTAP2
canonical CBOR encoding form, whitespace and line breaks are included
here for clarity and to match the [CDDL] presentation above:
A5
01 02

03 26

20 01

21 58 20 65eda5a12577c2bae829437fe338701a10aaa375e1bb5b5de108de439c08551d

22 58 20 1e52ed75701163f7f9e40ddf9f341b3dc9ba860af7e0ca7ca7e9eecd0084d19c

Below is an example of a COSE_Key-encoded 2048-bit RSA public key (see
[RFC8230] Section 4), to be used with the PS256 signature algorithm
(RSASSA-PSS with SHA-256, see [RFC8230] Section 2):
{
1: 3, ; kty: RSA key type
3: -37, ; alg: PS256
-1: n, ; n: RSA modulus n byte string 256 bytes in length
; e.g., in hex (middle bytes elided for brevity): DB5F651550...6
DC6548ACC3
-2: e ; e: RSA public exponent e byte string 3 bytes in length
; e.g., in hex: 010001
}

Below is an example of the same COSE_Key-encoded RSA public key as
above, to be used with the RS256 signature algorithm (RSASSA-PKCS1-v1_5
with SHA-256, see 11.3 COSE Algorithm Registrations):
{
1: 3, ; kty: RSA key type
3:-257, ; alg: RS256
-1: n, ; n: RSA modulus n byte string 256 bytes in length
; e.g., in hex (middle bytes elided for brevity): DB5F651550...6
DC6548ACC3
-2: e ; e: RSA public exponent e byte string 3 bytes in length
; e.g., in hex: 010001
}

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 $attStmtFormat
defined in 6.3.4 Generating an Attestation Object.
* Signing procedure: The signing procedure for computing an
attestation statement in this format given the public key
credential to be attested, the authenticator data structure
containing the authenticator data for the attestation, and the hash
of the serialized client data.
* Verification procedure: The procedure for verifying an attestation
statement, which takes the following verification procedure inputs:
+ attStmt: The attestation statement structure
+ authenticatorData: The authenticator data claimed to have been
used for the attestation
+ clientDataHash: The hash of the serialized client data
The procedure returns either:
+ An error indicating that the attestation is invalid, or
+ The attestation type, and the trust path. This attestation
trust path is either empty (in case of self attestation), an

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 $attStmtFormat
defined in 6.3.4 Generating an Attestation Object.
* Signing procedure: The signing procedure for computing an
attestation statement in this format given the public key
credential to be attested, the authenticator data structure
containing the authenticator data for the attestation, and the hash
of the serialized client data.
* Verification procedure: The procedure for verifying an attestation
statement, which takes the following verification procedure inputs:
+ attStmt: The attestation statement structure
+ authenticatorData: The authenticator data claimed to have been
used for the attestation
+ clientDataHash: The hash of the serialized client data
The procedure returns either:
+ An error indicating that the attestation is invalid, or
+ The attestation type, and the trust path. This attestation
trust path is either empty (in case of self attestation), an

identifier of a ECDAA-Issuer public key (in the case of ECDAA), or a set of X.509 certificates.

The initial list of specified attestation statement formats is in 8 Defined Attestation Statement Formats.

6.3.3. Attestation Types

WebAuthn supports multiple attestation types:

Basic Attestation
: In the case of basic attestation [UAFProtocol], the authenticator's attestation key pair is specific to an authenticator model. Thus, authenticators of the same model often share the same attestation key pair. See 6.3.5.1 Privacy for futher information.

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 generate multiple attestation key pairs and asks the Privacy CA to issue an attestation certificate for it. Using this approach, the Authenticator can limit the exposure of the endorsement key (which is a global correlation handle) to Privacy CA(s). Attestation keys can be requested for each public key credential individually.

  Note: This concept typically leads to multiple attestation certificates. The attestation certificate requested most recently is called "active".

Elliptic Curve based Direct Anonymous Attestation (ECDAA)
: In this case, the Authenticator receives direct anonymous attestation (DAA) credentials from a single DAA-Issuer. These DAA credentials are used along with blinding to sign the attested credential data. The concept of blinding avoids the DAA credentials being misused as global correlation handle. WebAuthn supports DAA using elliptic curve cryptography and bilinear pairings, called ECDAA (see [FIDOEcdaaAlgorithm]) in this specification. Consequently we denote the DAA-Issuer as ECDAA-Issuer (see [FIDOEcdaaAlgorithm]).

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.

identifier of an ECDAA-Issuer public key (in the case of ECDAA), or a set of X.509 certificates.

The initial list of specified attestation statement formats is in 8 Defined Attestation Statement Formats.

6.3.3. Attestation Types

WebAuthn supports multiple attestation types:

Basic Attestation (Basic)
: In the case of basic attestation [UAFProtocol], the authenticator's attestation key pair is specific to an authenticator model. Thus, authenticators of the same model often share the same attestation key pair. See 14.1 Attestation Privacy for further information.

Self Attestation (Self)
: 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.

Attestation CA (AttCA)
: In this case, an authenticator is based on a Trusted Platform Module (TPM) and holds an authenticator-specific "endorsement key" (EK). This key is used to securely communicate with a trusted third party, the Attestation CA [TCG-CMCProfile-AIKCertEnroll] (formerly known as a "Privacy CA"). The authenticator can generate multiple attestation identity key pairs (AIK) and requests an Attestation CA to issue an AIK certificate for each. Using this approach, such an authenticator can limit the exposure of the EK (which is a global correlation handle) to Attestation CA(s). AIKs can be requested for each authenticator-generated public key credential individually, and conveyed to Relying Parties as attestation certificates.

  Note: This concept typically leads to multiple attestation certificates. The attestation certificate requested most recently is called "active".

Elliptic Curve based Direct Anonymous Attestation (ECDAA)
: In this case, the Authenticator receives direct anonymous attestation (DAA) credentials from a single DAA-Issuer. These DAA credentials are used along with blinding to sign the attested credential data. The concept of blinding avoids the DAA credentials being misused as global correlation handle. WebAuthn supports DAA using elliptic curve cryptography and bilinear pairings, called ECDAA (see [FIDOEcdaaAlgorithm]) in this specification. Consequently we denote the DAA-Issuer as ECDAA-Issuer (see [FIDOEcdaaAlgorithm]).

No attestation statement (None)
: In this case, no attestation information is available.

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 identifier
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 key(s) (called Basic Attestation). This will anonymize the user at the risk of not being able to revoke a particular attestation key should its WebAuthn Authenticator be compromised.
* A WebAuthn Authenticator may be capable of dynamically generating different attestation keys (and requesting related certificates) per origin (following the Privacy CA approach). For example, a WebAuthn Authenticator can ship with a master attestation key (and certificate), and combined with a cloud operated privacy CA, can dynamically generate per origin attestation keys and attestation certificates.
* A WebAuthn Authenticator can implement 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.

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 identifier
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. Signature Formats for Packed Attestation, FIDO U2F Attestation, and Assertion Signatures

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 remember intermediate attestation CA certificates during Authenticator registration in order to un-register related public key credentials if the registration was performed after revocation of such certificates.

If an ECDAA attestation key has been compromised, it can be added to the RogueList (i.e., the list of revoked authenticators) maintained by the related ECDAA-Issuer. The Relying Party should verify whether an authenticator belongs to the RogueList when performing ECDAA-Verify (see section 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.

7. Relying Party Operations

Upon successful execution of create() or get(), the Relying Party's script receives a PublicKeyCredential containing an AuthenticatorAttestationResponse or AuthenticatorAssertionResponse structure, respectively, from the client. It must then deliver the contents of this structure to the Relying Party server, using methods outside the scope of this specification. This section describes the operations that the Relying Party must perform upon receipt of these structures.

7.1. Registering a new credential

When registering a new credential, represented by a AuthenticatorAttestationResponse structure, 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 type in C is the string webauthn.create.

* 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:
  ```
  30 44                   ; SEQUENCE (68 Bytes)
     02 20                ; INTEGER (32 Bytes)
     |  3d 46 28 7b 8c 6e 8c  26 1c 1b 88 f2 73 b0 9a
     |  32 a6 cf 28 09 fd 6e 30  d5 a7 9f 26 37 00 8f 54
     02 20                ; INTEGER (32 Bytes)
     |  4e 72 23 6e a3 90 a9 a1  7b cf 5f 7a 09 d6 3a b2
     |  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 AuthenticatorAttestationResponse or AuthenticatorAssertionResponse structure, respectively, from the client. It must then deliver the contents of this structure to the Relying Party server, using methods outside the scope of this specification. This section describes the operations that the Relying Party must perform upon receipt of these structures.

7.1. Registering a new credential

When registering a new credential, represented by an AuthenticatorAttestationResponse structure response 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.

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 tokenBindingId in C matches the Token Binding ID for the TLS connection over which the attestation was obtained.
6. Verify that the clientExtensions in C is a subset of the extensions requested by the RP and that the authenticatorExtensions in C is also a subset of the extensions requested by the RP.
7. Compute the hash of clientDataJSON using the algorithm identified by C.hashAlgorithm.

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 RP ID hash in authData is indeed the SHA-256 hash of the RP ID expected by the RP.
10. Determine the attestation statement format by performing an USASCII case-sensitive match on fmt against the set of supported WebAuthn Attestation Statement Format Identifier values. The up-to-date list of registered WebAuthn Attestation Statement Format Identifier values is maintained in the in the IANA registry of the same name [WebAuthn-Registries].
11. Verify that attStmt is a correct attestation statement, conveying a valid attestation signature, by using the attestation statement format fmt's verification procedure given attStmt, authData and the hash of the serialized client data computed in step 6.
Note: Each attestation statement format specifies its own verification procedure. See 8 Defined Attestation Statement Formats for the initially-defined formats, and [WebAuthn-Registries] for the up-to-date list.
12. If validation is successful, obtain a list of acceptable trust anchors (attestation root certificates or ECDAA-Issuer public keys) for that attestation type and attestation statement format fmt, from a trusted source or from policy. For example, the FIDO Metadata Service [FIDOMetadataService] provides one way to obtain such information, using the aaguid in the attestedCredentialData in authData.
13. Assess the attestation trustworthiness using the outputs of the verification procedure in step 10, as follows:
+ If self attestation was used, check if self attestation is acceptable under Relying Party policy.
+ If ECDAA was used, verify that the identifier of the

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) MUST be stripped.
2. Let C, the client data claimed as collected during the credential creation, be the result of running an implementation-specific JSON parser on JSONtext.
Note: C may be any implementation-specific data structure representation, as long as C's components are referenceable, as required by this algorithm.
3. Verify that the value of C.type is webauthn.create.
4. Verify that the value of C.challenge matches the challenge that was sent to the authenticator in the create() call.
5. Verify that the value of C.origin matches the Relying Party's origin.
6. Verify that the value of C.tokenBinding.status matches the state of Token Binding for the TLS connection over which the assertion was obtained. If Token Binding was used on that TLS connection, also verify that C.tokenBinding.id matches the base64url encoding of the Token Binding ID for the connection.
7. Compute the hash of response.clientDataJSON using SHA-256.

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 RP ID hash in authData is indeed the SHA-256 hash of the RP ID expected by the RP.
10. If user verification is required for this registration, verify that the User Verified bit of the flags in authData is set.
11. If user verification is not required for this registration, verify that the User Present bit of the flags in authData is set.
12. Verify that the values of the client extension outputs in clientExtensionResults and the authenticator extension outputs in the extensions in authData are as expected, considering the client extension input values that were given as the extensions option in the create() call. In particular, any extension identifier values in the clientExtensionResults and the extensions in authData MUST be also be present as extension identifier values in the extensions member of options, i.e., no extensions are present that were not requested. In the general case, the meaning of "are as expected" is specific to the Relying Party and which extensions are in use.
Note: Since all extensions are OPTIONAL for both the client and the authenticator, the Relying Party MUST be prepared to handle cases where none or not all of the requested extensions were acted upon.
13. Determine the attestation statement format by performing a USASCII case-sensitive match on fmt against the set of supported WebAuthn Attestation Statement Format Identifier values. The up-to-date list of registered WebAuthn Attestation Statement Format Identifier values is maintained in the in the IANA registry of the same name [WebAuthn-Registries].
14. Verify that attStmt is a correct attestation statement, conveying a valid attestation signature, by using the attestation statement format fmt's verification procedure given attStmt, authData and the hash of the serialized client data computed in step 7.
Note: Each attestation statement format specifies its own verification procedure. See 8 Defined Attestation Statement Formats for the initially-defined formats, and [WebAuthn-Registries] for the up-to-date list.
15. If validation is successful, obtain a list of acceptable trust anchors (attestation root certificates or ECDAA-Issuer public keys) for that attestation type and attestation statement format fmt, from a trusted source or from policy. For example, the FIDO Metadata Service [FIDOMetadataService] provides one way to obtain such information, using the aaguid in the attestedCredentialData in authData.
16. Assess the attestation trustworthiness using the outputs of the verification procedure in step 14, as follows:
+ If self attestation was used, check if self attestation is acceptable under Relying Party policy.
+ If ECDAA was used, verify that the identifier of the

ECDAA-Issuer public key used is included in the set of acceptable trust anchors obtained in step 11.

+ Otherwise, use the X.509 certificates returned by the verification procedure to verify that the attestation public key 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 credential public key but treat the credential as one with self attestation (see 6.3.3 Attestation Types). If doing so, the Relying Party is asserting there is no cryptographic proof that the public key credential has been generated by a particular authenticator model. See [FIDOSecRef] and [UAFProtocol] for a more detailed discussion.

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 follows:

1. Using credential's id attribute (or the corresponding rawId, if base64url encoding is inappropriate for your use case), look up the corresponding credential public key.
2. Let cData, aData and sig denote the value of credential's response's clientDataJSON, authenticatorData, and signature respectively.
3. Perform JSON deserialization on cData to extract the client data C used for the signature.
4. Verify that the type in C is the string webauthn.get.
5. Verify that the challenge member of C matches the challenge that was sent to the authenticator in the PublicKeyCredentialRequestOptions passed to the get() call.
6. Verify that the origin member of C matches the Relying Party's

ECDAA-Issuer public key used is included in the set of acceptable trust anchors obtained in step 15.

+ Otherwise, use the X.509 certificates returned by the verification procedure to verify that the attestation public key correctly chains up to an acceptable root certificate.

17. Check that the credentialId is not yet registered to any other user. If registration is requested for a credential that is already registered to a different user, the Relying Party SHOULD fail this registration ceremony, or it MAY decide to accept the registration, e.g. while deleting the older registration.

18. 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.

19. If the attestation statement attStmt successfully verified but is not trustworthy per step 16 above, the Relying Party SHOULD fail the registration ceremony.
NOTE: However, if permitted by policy, the Relying Party MAY register the credential ID and credential public key but treat the credential as one with self attestation (see 6.3.3 Attestation Types). If doing so, the Relying Party is asserting there is no cryptographic proof that the public key credential has been generated by a particular authenticator model. See [FIDOSecRef] and [UAFProtocol] for a more detailed discussion.

Verification of attestation objects requires that the Relying Party has a trusted method of determining acceptable trust anchors in step 15 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.

7.2. Verifying an authentication assertion

When verifying a given PublicKeyCredential structure (credential) and an AuthenticationExtensionsClientOutputs structure clientExtensionResults, as part of an authentication ceremony, the Relying Party MUST proceed as follows:

1. If the allowCredentials option was given when this authentication ceremony was initiated, verify that credential.id identifies one of the public key credentials that were listed in allowCredentials.
2. If credential.response.userHandle is present, verify that the user identified by this value is the owner of the public key credential identified by credential.id.
3. Using credential's id attribute (or the corresponding rawId, if base64url encoding is inappropriate for your use case), look up the corresponding credential public key.
4. Let cData, aData and sig denote the value of credential's response's clientDataJSON, authenticatorData, and signature respectively.
5. Let JSONtext be the result of running UTF-8 decode on the value of cData.
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) MUST be stripped.
6. Let C, the client data claimed as used for the signature, be the result of running an implementation-specific JSON parser on JSONtext.
Note: C may be any implementation-specific data structure

origin.

7. Verify that the tokenBindingId member of C (if present) matches the Token Binding ID for the TLS connection over which the signature was obtained.

8. Verify that the clientExtensions member of C is a subset of the extensions requested by the Relying Party and that the authenticatorExtensions in C is also a subset of the extensions requested by the Relying Party.

9. Verify that the rpIdHash in aData is the SHA-256 hash of the RP ID expected by the Relying Party.

10. Let hash be the result of computing a hash over the cData using the algorithm represented by the hashAlgorithm member of C.

11. Using the credential public key looked up in step 1, verify that

sig is a valid signature over the binary concatenation of aData and hash.

12. 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 substep:

+ If the signature counter value adata.signCount is

greater than the signature counter value stored in conjunction with credential's id attribute.
Update the stored signature counter value, associated with credential's id attribute, to be the value of adata.signCount.

less than or equal to the signature counter value stored in conjunction with credential's id attribute.
This is an signal that the authenticator may be cloned, i.e. at least two copies of the credential 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.

representation, as long as C's components are referenceable, as required by this algorithm.

7. Verify that the value of C.type is the string webauthn.get.

8. Verify that the value of C.challenge matches the challenge that was sent to the authenticator in the PublicKeyCredentialRequestOptions passed to the get() call.

9. Verify that the value of C.origin matches the Relying Party's origin.

10. Verify that the value of C.tokenBinding.status matches the state of Token Binding for the TLS connection over which the attestation was obtained. If Token Binding was used on that TLS connection, also verify that C.tokenBinding.id matches the base64url encoding of the Token Binding ID for the connection.

11. Verify that the rpIdHash in aData is the SHA-256 hash of the RP ID expected by the Relying Party.

12. If user verification is required for this assertion, verify that the User Verified bit of the flags in aData is set.

13. If user verification is not required for this assertion, verify that the User Present bit of the flags in aData is set.

14. Verify that the values of the client extension outputs in clientExtensionResults and the authenticator extension outputs in the extensions in authData are as expected, considering the client extension input values that were given as the extensions option in the get() call. In particular, any extension identifier values in the clientExtensionResults and the extensions in authData MUST be also be present as extension identifier values in the extensions member of options, i.e., no extensions are present that were not requested. In the general case, the meaning of "are as expected" is specific to the Relying Party and which extensions are in use.
Note: Since all extensions are OPTIONAL for both the client and the authenticator, the Relying Party MUST be prepared to handle cases where none or not all of the requested extensions were acted upon.

15. Let hash be the result of computing a hash over the cData using SHA-256.

16. Using the credential public key looked up in step 3, verify that sig is a valid signature over the binary concatenation of aData and hash.

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:

+ If the signature counter value adata.signCount is

greater than the signature counter value stored in conjunction with credential's id attribute.
Update the stored signature counter value, associated with credential's id attribute, to be the value of adata.signCount.

less than or equal to 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 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.

18. 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 a 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:

```
$$attStmtType //= (
                    fmt: "packed",
                    attStmt: packedStmtFormat
                )

packedStmtFormat = {
                     alg: COSEAlgorithmIdentifier,
                     sig: bytes,
                     x5c: [ attestnCert: bytes, * (caCert: bytes) ]
                   } //
                   {
                     alg: COSEAlgorithmIdentifier, (-260 for ED256 / -261
for ED512)
                     sig: bytes,
                     ecdaaKeyId: bytes
```


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.

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:

```
$$attStmtType //= (
                    fmt: "packed",
                    attStmt: packedStmtFormat
                )

packedStmtFormat = {
                     alg: COSEAlgorithmIdentifier,
                     sig: bytes,
                     x5c: [ attestnCert: bytes, * (caCert: bytes) ]
                   } //
                   {
                     alg: COSEAlgorithmIdentifier, (-260 for ED256 / -261
for ED512)
                     sig: bytes,
                     ecdaaKeyId: bytes
                   } //
                   {
```

```
3311                }
3312
3313        The semantics of the fields are as follows:
3314
3315        alg
3316            A COSEAlgorithmIdentifier containing the identifier of the
3317            algorithm used to generate the attestation signature.
3318
3319        sig
3320            A byte string containing the attestation signature.
3321
3322        x5c
3323            The elements of this array contain the attestation
3324            certificate and its certificate chain, each encoded in
3325            X.509 format. The attestation certificate must be the
3326            first element in the array.
3327
3328        ecdaaKeyId
3329            The identifier of the ECDAA-Issuer public key. This is the
3330            BigNumberToB encoding of the component "c" of the
3331            ECDAA-Issuer public key as defined section 3.3, step 3.5
3332            in [FIDOEcdaaAlgorithm].
3333
3334    Signing procedure
3335        The signing procedure for this attestation statement format is
3336        similar to the procedure for generating assertion signatures.
3337
3338        1. Let authenticatorData denote the authenticator data for the
3339           attestation, and let clientDataHash denote the hash of the
3340           serialized client data.
3341        2. If Basic or Privacy CA attestation is in use, the
3342           authenticator produces the sig by concatenating
3343           authenticatorData and clientDataHash, and signing the result
3344           using an attestation private key selected through an
3345           authenticator-specific mechanism. It sets x5c to the
3346           certificate chain of the attestation public key and alg to the
3347           algorithm of the attestation private key.
3348        3. If ECDAA is in use, the authenticator produces sig by
3349           concatenating authenticatorData and clientDataHash, and
3350           signing the result using ECDAA-Sign (see section 3.5 of
3351           [FIDOEcdaaAlgorithm]) after selecting an ECDAA-Issuer public
3352           key related to the ECDAA signature private key through an
3353           authenticator-specific mechanism (see [FIDOEcdaaAlgorithm]).
3354           It sets alg to the algorithm of the selected ECDAA-Issuer
3355           public key and ecdaaKeyId to the identifier of the
3356           ECDAA-Issuer public key (see above).
3357        4. If self attestation is in use, the authenticator produces sig
3358           by concatenating authenticatorData and clientDataHash, and
3359           signing the result using the credential private key. It sets
3360           alg to the algorithm of the credential private key, and omits
3361           the other fields.
3362
3363    Verification procedure
3364        Given the verification procedure inputs attStmt,
3365        authenticatorData and clientDataHash, the verification procedure
3366        is as follows:
3367
3368        1. Verify that attStmt is valid CBOR conforming to the syntax
3369           defined above, and perform CBOR decoding on it to extract the
3370           contained fields.
3371        2. If x5c is present, this indicates that the attestation type is
3372           not ECDAA. In this case:
3373             o Verify that sig is a valid signature over the
3374               concatenation of authenticatorData and clientDataHash
3375               using the attestation public key in x5c with the
3376               algorithm specified in alg.
3377             o Verify that x5c meets the requirements in 8.2.1 Packed
3378               attestation statement certificate requirements.
```

```
3625                    alg: COSEAlgorithmIdentifier
3626                    sig: bytes,
3627                }
3628
3629        The semantics of the fields are as follows:
3630
3631        alg
3632            A COSEAlgorithmIdentifier containing the identifier of the
3633            algorithm used to generate the attestation signature.
3634
3635        sig
3636            A byte string containing the attestation signature.
3637
3638        x5c
3639            The elements of this array contain the attestation
3640            certificate and its certificate chain, each encoded in
3641            X.509 format. The attestation certificate MUST be the
3642            first element in the array.
3643
3644        ecdaaKeyId
3645            The identifier of the ECDAA-Issuer public key. This is the
3646            BigNumberToB encoding of the component "c" of the
3647            ECDAA-Issuer public key as defined section 3.3, step 3.5
3648            in [FIDOEcdaaAlgorithm].
3649
3650    Signing procedure
3651        The signing procedure for this attestation statement format is
3652        similar to the procedure for generating assertion signatures.
3653
3654        1. Let authenticatorData denote the authenticator data for the
3655           attestation, and let clientDataHash denote the hash of the
3656           serialized client data.
3657        2. If Basic or AttCA attestation is in use, the authenticator
3658           produces the sig by concatenating authenticatorData and
3659           clientDataHash, and signing the result using an attestation
3660           private key selected through an authenticator-specific
3661           mechanism. It sets x5c to the certificate chain of the
3662           attestation public key and alg to the algorithm of the
3663           attestation private key.
3664        3. If ECDAA is in use, the authenticator produces sig by
3665           concatenating authenticatorData and clientDataHash, and
3666           signing the result using ECDAA-Sign (see section 3.5 of
3667           [FIDOEcdaaAlgorithm]) after selecting an ECDAA-Issuer public
3668           key related to the ECDAA signature private key through an
3669           authenticator-specific mechanism (see [FIDOEcdaaAlgorithm]).
3670           It sets alg to the algorithm of the selected ECDAA-Issuer
3671           public key and ecdaaKeyId to the identifier of the
3672           ECDAA-Issuer public key (see above).
3673        4. If self attestation is in use, the authenticator produces sig
3674           by concatenating authenticatorData and clientDataHash, and
3675           signing the result using the credential private key. It sets
3676           alg to the algorithm of the credential private key and omits
3677           the other fields.
3678
3679    Verification procedure
3680        Given the verification procedure inputs attStmt,
3681        authenticatorData and clientDataHash, the verification procedure
3682        is as follows:
3683
3684        1. Verify that attStmt is valid CBOR conforming to the syntax
3685           defined above and perform CBOR decoding on it to extract the
3686           contained fields.
3687        2. If x5c is present, this indicates that the attestation type is
3688           not ECDAA. In this case:
3689             o Verify that sig is a valid signature over the
3690               concatenation of authenticatorData and clientDataHash
3691               using the attestation public key in x5c with the
3692               algorithm specified in alg.
3693             o Verify that x5c meets the requirements in 8.2.1 Packed
3694               attestation statement certificate requirements.
```

```
3379          o If x5c contains an extension with OID 1 3 6 1 4 1 45724 1
3380            1 4 (id-fido-gen-ce-aaguid) verify that the value of this
3381            extension matches the aaguid in authenticatorData.
3382
3383          o If successful, return attestation type Basic and
3384            attestation trust path x5c.
3385      3. If ecdaaKeyId is present, then the attestation type is ECDAA.
3386        In this case:
3387          o Verify that sig is a valid signature over the
3388            concatenation of authenticatorData and clientDataHash
3389            using ECDAA-Verify with ECDAA-Issuer public key
3390            identified by ecdaaKeyId (see [FIDOEcdaaAlgorithm]).
3391          o If successful, return attestation type ECDAA and
3392            attestation trust path ecdaaKeyId.
3393      4. If neither x5c nor ecdaaKeyId is present, self attestation is
3394        in use.
3395          o Validate that alg matches the algorithm of the
3396            credentialPublicKey in authenticatorData.
3397          o Verify that sig is a valid signature over the
3398            concatenation of authenticatorData and clientDataHash
3399            using the credential public key with alg.
3400          o If successful, return attestation type Self and empty
3401            attestation trust path.
3402
3403    8.2.1. Packed attestation statement certificate requirements
3404
3405  The attestation certificate MUST have the following fields/extensions:
3406    * Version must be set to 3.
3407
3408    * Subject field MUST be set to:
3409
3410      Subject-C
3411            Country where the Authenticator vendor is incorporated
3412
3413      Subject-O
3414            Legal name of the Authenticator vendor
3415
3416      Subject-OU
3417            Authenticator Attestation
3418
3419      Subject-CN
3420            No stipulation.
3421
3422    * If the related attestation root certificate is used for multiple
3423      authenticator models, the Extension OID 1 3 6 1 4 1 45724 1 1 4
3424      (id-fido-gen-ce-aaguid) MUST be present, containing the AAGUID as
3425      value.
3426
3427    * The Basic Constraints extension MUST have the CA component set to
3428      false
3429    * An Authority Information Access (AIA) extension with entry
3430      id-ad-ocsp and a CRL Distribution Point extension [RFC5280] are
3431      both optional as the status of many attestation certificates is
3432      available through authenticator metadata services. See, for
3433      example, the FIDO Metadata Service [FIDOMetadataService].
3434
3435  8.3. TPM Attestation Statement Format
```

```
3695          o If x5c contains an extension with OID
3696            1.3.6.1.4.1.45724.1.1.4 (id-fido-gen-ce-aaguid) verify
3697            that the value of this extension matches the aaguid in
3698            authenticatorData.
3699          o If successful, return attestation type Basic and
3700            attestation trust path x5c.
3701      3. If ecdaaKeyId is present, then the attestation type is ECDAA.
3702        In this case:
3703          o Verify that sig is a valid signature over the
3704            concatenation of authenticatorData and clientDataHash
3705            using ECDAA-Verify with ECDAA-Issuer public key
3706            identified by ecdaaKeyId (see [FIDOEcdaaAlgorithm]).
3707          o If successful, return attestation type ECDAA and
3708            attestation trust path ecdaaKeyId.
3709      4. If neither x5c nor ecdaaKeyId is present, self attestation is
3710        in use.
3711          o Validate that alg matches the algorithm of the
3712            credentialPublicKey in authenticatorData.
3713          o Verify that sig is a valid signature over the
3714            concatenation of authenticatorData and clientDataHash
3715            using the credential public key with alg.
3716          o If successful, return attestation type Self and empty
3717            attestation trust path.
3718
3719    8.2.1. Packed attestation statement certificate requirements
3720
3721  The attestation certificate MUST have the following fields/extensions:
3722    * Version MUST be set to 3 (which is indicated by an ASN.1 INTEGER
3723      with value 2).
3724    * Subject field MUST be set to:
3725
3726      Subject-C
3727            ISO 3166 code specifying the country where the
3728            Authenticator vendor is incorporated (PrintableString)
3729
3730      Subject-O
3731            Legal name of the Authenticator vendor (UTF8String)
3732
3733      Subject-OU
3734            Literal string "Authenticator Attestation" (UTF8String)
3735
3736      Subject-CN
3737            A UTF8String of the vendor's choosing
3738
3739    * If the related attestation root certificate is used for multiple
3740      authenticator models, the Extension OID 1.3.6.1.4.1.45724.1.1.4
3741      (id-fido-gen-ce-aaguid) MUST be present, containing the AAGUID as a
3742      16-byte OCTET STRING. The extension MUST NOT be marked as critical.
3743      Note that an X.509 Extension encodes the DER-encoding of the value
3744      in an OCTET STRING. Thus, the AAGUID must be wrapped in two OCTET
3745      STRINGS to be valid. Here is a sample, encoded Extension structure:
3746  30 21                       -- SEQUENCE
3747    06 0b 2b 06 01 04 01 82 e5 1c 01 01 04  -- 1.3.6.1.4.1.45724.1.1.4
3748    04 12                     -- OCTET STRING
3749      04 10                   -- OCTET STRING
3750        cd 8c 39 5c 26 ed ee de         -- AAGUID
3751        65 3b 00 79 7d 03 ca 3c
3752
3753    * The Basic Constraints extension MUST have the CA component set to
3754      false.
3755    * An Authority Information Access (AIA) extension with entry
3756      id-ad-ocsp and a CRL Distribution Point extension [RFC5280] are
3757      both OPTIONAL as the status of many attestation certificates is
3758      available through authenticator metadata services. See, for
3759      example, the FIDO Metadata Service [FIDOMetadataService].
3760
3761  8.3. TPM Attestation Statement Format
3762
3763  This attestation statement format is generally used by authenticators
3764  that use a Trusted Platform Module as their cryptographic engine.
```

Attestation statement format identifier
tpm

Attestation types supported
Privacy CA, ECDAA

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

```
$$attStmtType // = (
                    fmt: "tpm",
                    attStmt: tpmStmtFormat
                )

tpmStmtFormat = {
                    ver: "2.0",
                    (
                        alg: COSEAlgorithmIdentifier,
                        x5c: [ aikCert: bytes, * (caCert: bytes) ]
                    ) //
                    (
                        alg:  COSEAlgorithmIdentifier, (-260 for ED256 / -26
1 for ED512)
                        ecdaaKeyId: bytes
                    ),
                    sig: bytes,
                    certInfo: bytes,
                    pubArea: bytes
                }
```

The semantics of the above fields are as follows:

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

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

x5c
The AIK certificate used for the attestation and its certificate chain, in X.509 encoding.

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

sig
The attestation signature, in the form of a TPMT_SIGNATURE structure as specified in [TPMv2-Part2] section 11.3.4.

certInfo
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

Attestation statement format identifier
tpm

Attestation types supported
AttCA, ECDAA

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

```
$$attStmtType // = (
                    fmt: "tpm",
                    attStmt: tpmStmtFormat
                )

tpmStmtFormat = {
                    ver: "2.0",
                    (
                        alg: COSEAlgorithmIdentifier,
                        x5c: [ aikCert: bytes, * (caCert: bytes) ]
                    ) //
                    (
                        alg:  COSEAlgorithmIdentifier, (-260 for ED256 / -26
1 for ED512)
                        ecdaaKeyId: bytes
                    ),
                    sig: bytes,
                    certInfo: bytes,
                    pubArea: bytes
                }
```

The semantics of the above fields are as follows:

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

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

x5c
The AIK certificate used for the attestation and its certificate chain, in X.509 encoding.

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

sig
The attestation signature, in the form of a TPMT_SIGNATURE structure as specified in [TPMv2-Part2] section 11.3.4.

certInfo
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-Part3] Section 18.2, using the attestation 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 public key, the certInfo field to the output parameter of the same name, and the sig field to the signature obtained from the above procedure.

Verification procedure

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 specified by the parameters and unique fields of pubArea is identical to the credentialPublicKey in the attestedCredentialData in authenticatorData.

Concatenate authenticatorData and clientDataHash to form attToBeSigned.

Validate that certInfo is valid:

+ Verify that magic is set to TPM_GENERATED_VALUE.
+ Verify that type is set to TPM_ST_ATTEST_CERTIFY.
+ Verify that extraData is set to the hash of attToBeSigned using the hash algorithm employed in "alg".
+ Verify that attested contains a TPMS_CERTIFY_INFO structure, whose name field contains a valid Name for pubArea, as computed using the algorithm in the 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 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.
+ If x5c contains an extension with OID 1 3 6 1 4 1 45724 1 1 4 (id-fido-gen-ce-aaguid) verify that the value of this extension matches the aaguid in authenticatorData.
+ If successful, return attestation type Privacy CA and attestation trust path x5c.

If ecdaaKeyId is present, then the attestation type is ECDAA.

+ Perform ECDAA-Verify on sig to verify that it is a valid signature over certInfo (see [FIDOEcdaaAlgorithm]).
+ If successful, return attestation type ECDAA and the identifier of the ECDAA-Issuer public key ecdaaKeyId.

8.3.1. TPM attestation statement certificate requirements

TPM attestation certificate MUST have the following fields/extensions:

attToBeSigned.

Generate a signature using the procedure specified in [TPMv2-Part3] Section 18.2, using the attestation 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 public key, the certInfo field to the output parameter of the same name, and the sig field to the signature obtained from the above procedure.

Verification procedure

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 specified by the parameters and unique fields of pubArea is identical to the credentialPublicKey in the attestedCredentialData in authenticatorData.

Concatenate authenticatorData and clientDataHash to form attToBeSigned.

Validate that certInfo is valid:

+ Verify that magic is set to TPM_GENERATED_VALUE.
+ Verify that type is set to TPM_ST_ATTEST_CERTIFY.
+ Verify that extraData is set to the hash of attToBeSigned using the hash algorithm employed in "alg".
+ Verify that attested contains a TPMS_CERTIFY_INFO structure as specified in [TPMv2-Part2] section 10.12.3, 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.
+ Note that the remaining fields in the "Standard Attestation Structure" [TPMv2-Part1] section 31.2, i.e., qualifiedSigner, clockInfo and firmwareVersion are ignored. These fields MAY be used as an input to risk engines.

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

+ Verify the sig is a valid signature over certInfo using the attestation public key in x5c with the algorithm specified in alg.
+ Verify that x5c meets the requirements in 8.3.1 TPM attestation statement certificate requirements.
+ If x5c contains an extension with OID 1 3 6 1 4 1 45724 1 1 4 (id-fido-gen-ce-aaguid) verify that the value of this extension matches the aaguid in authenticatorData.
+ If successful, return attestation type AttCA and attestation trust path x5c.

If ecdaaKeyId is present, then the attestation type is ECDAA.

+ Perform ECDAA-Verify on sig to verify that it is a valid signature over certInfo (see [FIDOEcdaaAlgorithm]).
+ If successful, return attestation type ECDAA and the identifier of the ECDAA-Issuer public key ecdaaKeyId.

8.3.1. TPM attestation statement certificate requirements

TPM attestation certificate MUST have the following fields/extensions:

* Version must be set to 3.
* Subject field MUST be set to empty.
* The Subject Alternative Name extension must be set as defined in [TPMv2-EK-Profile] section 3.2.9.
* The Extended Key Usage extension MUST contain the "joint-iso-itu-t(2) internationalorganizations(23) 133 tcg-kp(8) tcg-kp-AIKCertificate(3)" OID.
* The Basic Constraints extension MUST have the CA component set to false.
* An Authority Information Access (AIA) extension with entry id-ad-ocsp and a CRL Distribution Point extension [RFC5280] are both optional as the status of many attestation certificates is available through metadata services. See, for example, the FIDO Metadata Service [FIDOMetadataService].

8.4. Android Key Attestation Statement Format

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

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:

```
$$attStmtType //= (
                    fmt: "android-key",
                    attStmt: androidStmtFormat
                  )

androidStmtFormat = {
                      alg: 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 serialized client data.

  Request an Android Key Attestation by calling "keyStore.getCertificateChain(myKeyUUID)") 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:

* Version MUST be set to 3.
* Subject field MUST be set to empty.
* The Subject Alternative Name extension MUST be set as defined in [TPMv2-EK-Profile] section 3.2.9.
* The Extended Key Usage extension MUST contain the "joint-iso-itu-t(2) internationalorganizations(23) 133 tcg-kp(8) tcg-kp-AIKCertificate(3)" OID.
* The Basic Constraints extension MUST have the CA component set to false.
* An Authority Information Access (AIA) extension with entry id-ad-ocsp and a CRL Distribution Point extension [RFC5280] are both OPTIONAL as the status of many attestation certificates is available through metadata services. See, for example, the FIDO Metadata Service [FIDOMetadataService].

8.4. Android Key Attestation Statement Format

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

Attestation statement format identifier
: android-key

Attestation types supported
: Basic

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

```
$$attStmtType //= (
                    fmt: "android-key",
                    attStmt: androidStmtFormat
                  )

androidStmtFormat = {
                      alg: 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 serialized client data.

  Request an Android Key Attestation by calling keyStore.getCertificateChain(myKeyUUID) 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 the concatenation of authenticatorData and
      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:

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

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

The semantics of the above fields are as follows:

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

response
: The 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.

```
+ 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 sig is a valid signature over the concatenation of
  authenticatorData and clientDataHash using the public key in
  the first certificate in x5c with the algorithm specified in
  alg.
+ Verify that the public key in the first certificate in in x5c
  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 PublicKeyCredential 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 x5c.
```

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

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

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

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

The semantics of the above fields are as follows:

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

response
: The 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.

Signing procedure

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

Concatenate authenticatorData and clientDataHash to form attToBeSigned.

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

Verification procedure

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

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].

Attestation statement format identifier

fido-u2f

Attestation types supported

Basic Attestation, Self Attestation, Privacy CA

Syntax

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

```
$$attStmtType //= (
                    fmt: "fido-u2f",
                    attStmt: u2fStmtFormat
                  )

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

The semantics of the above fields are as follows:

x5c

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

sig

The attestation signature. The signature was calculated over the (raw) U2F registration response message

Signing procedure

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

Concatenate authenticatorData and clientDataHash to form attToBeSigned.

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

Verification procedure

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

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].

Attestation statement format identifier

fido-u2f

Attestation types supported

Basic, AttCA

Syntax

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

```
$$attStmtType //= (
                    fmt: "fido-u2f",
                    attStmt: u2fStmtFormat
                  )

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

The semantics of the above fields are as follows:

x5c

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

sig

The attestation signature. The signature was calculated over the (raw) U2F registration response message

[FIDO-U2F-Message-Formats] received by the platform from the authenticator.

Signing procedure

If the credential public key of the given credential is not of algorithm -7 ("ES256"), stop and return an error. Otherwise, let authenticatorData denote the authenticator data for the attestation, and let clientDataHash denote the hash of the serialized client data.

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

Generate a 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 user public key, key handle, and attestation certificates) as sig and set the attestation certificates of the attestation public key as x5c.

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 x5c. Let certificate public key be the public key conveyed by attCert. 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.
3. Extract the claimed rpIdHash 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.
5. Convert the COSE_KEY formatted credentialPublicKey (see Section 7 of [RFC8152]) to CTAP1/U2F public Key format [FIDO-CTAP].
   - o Let publicKeyU2F represent the result of the conversion operation and set its first byte to 0x04. Note: This signifies uncompressed ECC key format.
   - o Extract the value corresponding to the "-2" key (representing x coordinate) from credentialPublicKey, 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.
   - o 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.
6. Let verificationData be the concatenation of (0x00 || rpIdHash || tbsHash || credentialId || publicKeyU2F) (see Section 4.3 of [FIDO-U2F-Message-Formats]).
7. Verify the sig using verificationData and certificate public key per [SEC1].
8. If successful, return attestation type Basic with the attestation trust path set to x5c.

[FIDO-U2F-Message-Formats] received by the platform from the authenticator.

Signing procedure

If the credential public key of the given credential is not of algorithm -7 ("ES256"), stop and return an error. Otherwise, let authenticatorData denote the authenticator data for the attestation, and let clientDataHash denote the hash of the serialized client data. (Since SHA-256 is used to hash the serialized client data, clientDataHash will be 32 bytes long.)

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 clientDataHash, 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 user public key, key handle, and attestation certificates) as sig and set the attestation certificates of the attestation public key as x5c.

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 the value of the first element of x5c. Let certificate public key be the public key conveyed by attCert. 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.
3. Extract the claimed rpIdHash from authenticatorData, and the claimed credentialId and credentialPublicKey from authenticatorData.attestedCredentialData.
4. Convert the COSE_KEY formatted credentialPublicKey (see Section 7 of [RFC8152]) to CTAP1/U2F public Key format [FIDO-CTAP].
   - o Let publicKeyU2F represent the result of the conversion operation and set its first byte to 0x04. Note: This signifies uncompressed ECC key format.
   - o Extract the value corresponding to the "-2" key (representing x coordinate) from credentialPublicKey, 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.
   - o 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.
5. Let verificationData be the concatenation of (0x00 || rpIdHash || clientDataHash || credentialId || publicKeyU2F) (see Section 4.3 of [FIDO-U2F-Message-Formats]).
6. Verify the sig using verificationData and certificate public key per [SEC1].
7. If successful, return attestation type Basic with the attestation trust path set to x5c.

8.7. None Attestation Statement Format

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 invoves 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 authentication extensions.
* Authenticator extension processing for registration extensions and authentication extensions.

For authenticator extensions, as part of the client extension

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).

Attestation statement format identifier
none

Attestation types supported
None

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

```
$$attStmtType //= (
                    fmt: "none",
                    attStmt: emptyMap
                  )

emptyMap = {}
```

Signing 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 authentication extensions.
* Authenticator extension processing for registration extensions and authentication extensions.

For authenticator extensions, as part of the client extension

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

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

Clients wishing to support the widest possible range of extensions may choose to pass through any extensions that they do not recognize to authenticators, generating the authenticator extension input by simply encoding the client extension input in CBOR. All WebAuthn extensions MUST be defined in such a way that this implementation choice does not endanger the user's security or privacy. For instance, if an extension requires client processing, it could be defined in a manner that ensures such a nave 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.

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

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

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

Clients wishing to support the widest possible range of extensions MAY choose to pass through any extensions that they do not recognize to authenticators, generating the authenticator extension input by simply encoding the client extension input in CBOR. All WebAuthn extensions MUST be defined in such a way that this implementation choice does not endanger the user's security or privacy. For instance, if an extension requires client processing, it could be defined in a manner that ensures such a nave 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 inputs. When 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 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 done, 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 through that they do not recognize, the CBOR values in the authenticator extension outputs are converted to JavaScript values in the client extension outputs. When the CBOR value is a byte string, it is converted to a JavaScript %ArrayBuffer% (rather than a base64url-encoded string). Otherwise, when the CBOR type corresponds to a JSON type, the conversion is done using the rules defined in Section 4.1 of [RFC7049] (Converting from CBOR to JSON), but producing outputs of JavaScript type values rather than outputs of JSON type values.

Note that some clients may choose to implement this pass-through capability under a feature flag. Supporting this capability can facilitate innovation, allowing authenticators to experiment with new extensions and Relying Parties to use them before there is explicit support for them in clients.

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

registered WebAuthn Extensions.

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

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

All extension identifiers MUST be a maximum of 32 octets in length and MUST consist only of printable USASCII characters, excluding backslash and doublequote, i.e., VCHAR as defined in [RFC5234] but without %x22 and %x5c. Implementations MUST match WebAuthn extension identifiers in a case-sensitive fashion.

Extensions that may exist in multiple versions should take care to include a version in their identifier. In effect, different versions are thus treated as different extensions, e.g., myCompany_extension_01

10 Defined Extensions defines an initial set of extensions and their identifiers. See the IANA "WebAuthn Extension Identifier" registry established by [WebAuthn-Registries] 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 honored by the client. Similarly, any extension that requires authenticator processing MUST return an authenticator extension output to let the Relying Party know that the extension was honored by the authenticator. If an extension does not otherwise require any result values, it SHOULD be defined as returning a JSON Boolean client extension output result, set to true to signify 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({
```


registered WebAuthn Extensions.

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

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

All extension identifiers MUST be a maximum of 32 octets in length and MUST consist only of printable USASCII characters, excluding backslash and doublequote, i.e., VCHAR as defined in [RFC5234] but without %x22 and %x5c. Implementations MUST match WebAuthn extension identifiers in a case-sensitive fashion.

Extensions that may exist in multiple versions should take care to include a version in their identifier. In effect, different versions are thus treated as different extensions, e.g., myCompany_extension_01

10 Defined Extensions defines an initial set of extensions and their identifiers. See the IANA "WebAuthn Extension Identifier" registry established by [WebAuthn-Registries] 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 honored by the client. Similarly, any extension that requires authenticator processing MUST return an authenticator extension output to let the Relying Party know that the extension was honored by the authenticator. If an extension does not otherwise require any result values, it SHOULD be defined as returning a JSON Boolean client extension output result, set to true to signify 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
        }
    }
});
```

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 authenticator extension input. Extensions that have authenticator processing MUST specify the method of computing the authenticator extension input from the client extension input. For extensions that do not require 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 generation 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.


```
    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 authenticator extension input. Extensions that have authenticator processing MUST specify the method of computing the authenticator extension input from the client extension input. For extensions that do not require 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 generation 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 parameter of the authenticatorMakeCredential and authenticatorGetAssertion operations. The extensions parameter is a CBOR map where each key is an extension identifier and the corresponding value is the authenticator extension input for that 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({
      publicKey: {
         // The challenge must be produced by the server, see the Security Co
nsiderations
         challenge: new Uint8Array([11,103,35 /* 29 more random bytes generat
ed by the server */]),
         allowCredentials: [], /* Empty filter */
         extensions: { 'webauthnExample_geo': true }
      }
   });
```

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

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

```
81 (hex)                          -- Flags, ED and UP both set.
20 05 58 1F                        -- Signature counter
A1                                -- CBOR map of one element
   73                             -- Key 1: CBOR text string of 19 byt
es
      77 65 62 61 75 74 68 6E 45 78 61
      6D 70 6C 65 5F 67 65 6F          -- "webauthnExample_geo" [=UTF-8 enc
oded=] string
   82                             -- Value 1: CBOR array of two elemen
ts
      FA 42 82 1E B3                -- Element 1: Latitude as CBOR encod
ed float
      FA C1 5F E3 7F                -- Element 2: Longitude as CBOR enco
ded float
```

The extension defines the client extension output to be the geolocation information, if known, as a GeoJSON [GeoJSON] point. The client constructs the following client data:

```
{
   ...,
   'extensions': {
      'webauthnExample_geo': {
         'type': 'Point',
         'coordinates': [65.059962, -13.993041]
      }
   }
}
```

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.

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.

Extension identifier
: appid

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

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

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.

10.2. Simple Transaction Authorization Extension (txAuthSimple)

This registration extension and authentication extension allows for a

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.

Extension identifier
: appid

Client extension input
: A single USVString specifying a FIDO AppID.

```
partial dictionary AuthenticationExtensionsClientInputs {
  USVString 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. 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.

```
partial dictionary AuthenticationExtensionsClientOutputs {
  boolean appid;
};
```

Authenticator extension input
: None.

Authenticator extension processing
: None.

Authenticator extension output
: None.

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.

Extension identifier
txAuthSimple

Client extension input
A single JSON string prompt.

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

Client extension output
Returns the authenticator extension output string UTF-8 decoded into a JSON string

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

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

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

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.

Extension identifier
txAuthGeneric

Client extension input
A CBOR map defined as follows:

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

Client extension processing
None, except creating the authenticator extension input from the

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

Extension identifier
txAuthSimple

Client extension input
A single USVString prompt.

```
partial dictionary AuthenticationExtensionsClientInputs {
  USVString txAuthSimple;
};
```

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

Client extension output
Returns the authenticator extension output string UTF-8 decoded into a USVString.

```
partial dictionary AuthenticationExtensionsClientOutputs {
  USVString txAuthSimple;
};
```

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

CDDL:
txAuthSimpleInput = (tstr)

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

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

CDDL:
txAuthSimpleOutput = (tstr)

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.

Extension identifier
txAuthGeneric

Client extension input
A JavaScript object defined as follows:

```
dictionary txAuthGenericArg {
   required USVString contentType;   // MIME-Type of the content, e.g., "image/png"
   required ArrayBuffer content;
};

partial dictionary AuthenticationExtensionsClientInputs {
  txAuthGenericArg txAuthGeneric;
};
```

Client extension processing
None, except creating the authenticator extension input from the

client extension input.

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

Authenticator extension input
The client extension input encoded as a CBOR map.

Authenticator extension processing
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.

Authenticator extension output
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
A sequence of AAGUIDs:

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

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

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

```
typedef BufferSource      AAGUID;
```

Client extension processing
This extension can only be used during create(). If the client supports the Authenticator Selection Extension, it MUST use the first available authenticator whose AAGUID is present in the AuthenticatorSelectionList. If none of the available authenticators match a provided AAGUID, the client MUST select an authenticator from among the available authenticators to generate the credential.

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

client extension input.

Client extension output
Returns the authenticator extension output value as an ArrayBuffer.

```
partial dictionary AuthenticationExtensionsClientOutputs {
  ArrayBuffer txAuthGeneric;
};
```

Authenticator extension input
The client extension input encoded as a CBOR map.

Authenticator extension processing
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.

Authenticator extension output
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
A sequence of AAGUIDs:

```
typedef sequence<AAGUID> AuthenticatorSelectionList;

partial dictionary AuthenticationExtensionsClientInputs {
  AuthenticatorSelectionList authnSel;
};
```

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

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

```
typedef BufferSource      AAGUID;
```

Client extension processing
This extension can only be used during create(). If the client supports the Authenticator Selection Extension, it MUST use the first available authenticator whose AAGUID is present in the AuthenticatorSelectionList. If none of the available authenticators match a provided AAGUID, the client MUST select an authenticator from among the available authenticators to generate the credential.

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

```
partial dictionary AuthenticationExtensionsClientOutputs {
  boolean authnSel;
};
```

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.

Extension identifier
exts

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 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 21).

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)

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

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.

Extension identifier
exts

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

```
partial dictionary AuthenticationExtensionsClientInputs {
  boolean exts;
};
```

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

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

```
typedef sequence<USVString> AuthenticationExtensionsSupported;

partial dictionary AuthenticationExtensionsClientOutputs {
  AuthenticationExtensionsSupported exts;
};
```

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 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)

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.

```
partial dictionary AuthenticationExtensionsClientInputs {
  boolean uvi;
};
```

Client extension processing
None, except creating the authenticator extension input from the

client extension input.

Client extension output
Returns 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 a user verification index indicating the method used by the user to authorize the operation, as defined below. This extension can be added to attestation objects and assertions.

Authenticator extension output
The user verification index (UVI) is a value uniquely identifying a user verification data record. The UVI is encoded as CBOR byte string (type 0x58). Each UVI value MUST be specific to the related key (in order to provide unlinkability). It also must contain sufficient entropy that makes guessing impractical. UVI values MUST NOT be reused by the Authenticator (for other biometric data or users).

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

As an example, the UVI could be computed as SHA256(KeyID || SHA256(rawUVI)), where || 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)
81                               -- UP and ED set
00 00 00 01                         -- (initial) signature counter
...                             -- all public key alg etc.
A1                               -- extension: CBOR map of one elemen
t
   63                            -- Key 1: CBOR text string of 3 byte
s
      75 76 69                      -- "uvi" [=UTF-8 encoded=] string
   58 20                         -- Value 1: CBOR byte string with 0x
20 bytes
      00 43 B8 E3 BE 27 95 8C          -- the UVI value itself
      28 D5 74 BF 46 8A 85 CF
      46 9A 14 F0 E5 16 69 31
      DA 4B CF FF C1 BB 11 32
      82
```

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

client extension input.

Client extension output
Returns the authenticator extension output as an ArrayBuffer.

```
partial dictionary AuthenticationExtensionsClientOutputs {
  ArrayBuffer uvi;
};
```

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

Authenticator extension output
The user verification index (UVI) is a value uniquely identifying a user verification data record. The UVI is encoded as CBOR byte string (type 0x58). Each UVI value MUST be specific to the related key (in order to provide unlinkability). It also MUST contain sufficient entropy that makes guessing impractical. UVI values MUST NOT be reused by the Authenticator (for other biometric data or users).

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

As an example, the UVI could be computed as SHA256(KeyID || SHA256(rawUVI)), where || 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)
81                               -- UP and ED set
00 00 00 01                         -- (initial) signature counter
...                             -- all public key alg etc.
A1                               -- extension: CBOR map of one elemen
t
   63                            -- Key 1: CBOR text string of 3 byte
s
      75 76 69                      -- "uvi" [=UTF-8 encoded=] string
   58 20                         -- Value 1: CBOR byte string with 0x
20 bytes
      43 B8 E3 BE 27 95 8C 28          -- the UVI value itself
      D5 74 BF 46 8A 85 CF 46
      9A 14 F0 E5 16 69 31 DA
      4B CF FF C1 BB 11 32 82
```

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 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 defined in The W3C Geolocation API Specification.

```
...                                 -- [=RP ID=] hash (32 bytes)
81                                   -- UP and ED set
00 00 00 01                             -- (initial) signature counter
...                                 -- all public key alg etc.
A1                                   -- extension: CBOR map of one elemen
t
   63                               -- Value 1: CBOR text string of 3 by
tes
      6C 6F 63                          -- "loc" [=UTF-8 encoded=] string
   86                               -- Value 2: array of 6 elements
      68                    -- Element 1:  CBOR text string of 8 bytes
        6C 61 74 69 74 75 64 65         -- "latitude" [=UTF-8 encoded=] stri
ng
      FB ...                -- Element 2:  Latitude as CBOR encoded double-p
recision float
      69                    -- Element 3:  CBOR text string of 9 bytes
        6C 6F 6E 67 69 74 75 64 65      -- "longitude" [=UTF-8 encoded=] str
ing
      FB ...                -- Element 4:  Longitude as CBOR encoded double-
precision float
      68                    -- Element 5:  CBOR text string of 8 bytes
        61 6C 74 69 74 75 64 65         -- "altitude" [=UTF-8 encoded=] stri
ng
      FB ...                -- Element 6:  Altitude as CBOR encoded double-p
```

```
partial dictionary AuthenticationExtensionsClientInputs {
  boolean loc;
};
```

Client extension processing
: None, except creating the authenticator extension input from the client extension input.

Client extension output
: Returns a JavaScript object that encodes the location information in the authenticator extension output as a Coordinates value, as defined by [Geolocation-API].

```
partial dictionary AuthenticationExtensionsClientOutputs {
  Coordinates loc;
};
```

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
: A [Geolocation-API] Coordinates record encoded as a CBOR map. 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.

recision float

10.8. User Verification Method Extension (uvm)

This registration extension and authentication extension enables use of a user verification method.

Extension identifier
: uvm

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

Client extension processing
: None, except creating the authenticator extension input from the client extension input.

Client extension output
: Returns a JSON array of 3-element arrays of numbers that encodes the factors in 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 one or more user verification methods indicating the method(s) used by the user to authorize the operation, as defined below. This extension can be added to attestation objects and assertions.

Authenticator extension output
: Authenticators can report up to 3 different user verification methods (factors) used in a single authentication instance, using the CBOR syntax defined below:

```
uvmFormat = [ 1*3 uvmEntry ]
uvmEntry = [
            userVerificationMethod: uint .size 4,
            keyProtectionType: uint .size 2,
            matcherProtectionType: uint .size 2
          ]
```

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

10.8. User Verification Method Extension (uvm)

This registration extension and authentication extension enables use of a user verification method.

Extension identifier
: uvm

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

```
partial dictionary AuthenticationExtensionsClientInputs {
  boolean uvm;
};
```

Client extension processing
: None, except creating the authenticator extension input from the client extension input.

Client extension output
: Returns a JSON array of 3-element arrays of numbers that encodes the factors in the authenticator extension output.

```
typedef sequence<unsigned long> UvmEntry;
typedef sequence<UvmEntry> UvmEntries;

partial dictionary AuthenticationExtensionsClientOutputs {
  UvmEntries uvm;
};
```

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 one or more user verification methods indicating the method(s) used by the user to authorize the operation, as defined below. This extension can be added to attestation objects and assertions.

Authenticator extension output
: Authenticators can report up to 3 different user verification methods (factors) used in a single authentication instance, using the CBOR syntax defined below:

```
uvmFormat = [ 1*3 uvmEntry ]
uvmEntry = [
            userVerificationMethod: uint .size 4,
            keyProtectionType: uint .size 2,
            matcherProtectionType: uint .size 2
          ]
```

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:

...             -- [=RP ID=] hash (32 bytes)
81               -- UP and ED set
00 00 00 01          -- (initial) signature counter
...             -- 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
 Fingerprint
         04        -- Subitem 2: CBOR short for Key Protection Type TEE
         02        -- Subitem 3: CBOR short for Matcher Protection Type TE
E
      83          -- Item 2: CBOR array of length 3
         04        -- Subitem 1: CBOR integer for User Verification Method
 Passcode
         01        -- Subitem 2: CBOR short for Key Protection Type Softwa
re
         01        -- Subitem 3: CBOR short for Matcher Protection Type So
ftware
```



```
            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:

...             -- [=RP ID=] hash (32 bytes)
81               -- UP and ED set
00 00 00 01          -- (initial) signature counter
...             -- 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
 Fingerprint
         04        -- Subitem 2: CBOR short for Key Protection Type TEE
         02        -- Subitem 3: CBOR short for Matcher Protection Type TE
E
      83          -- Item 2: CBOR array of length 3
         04        -- Subitem 1: CBOR integer for User Verification Method
 Passcode
         01        -- Subitem 2: CBOR short for Key Protection Type Softwa
re
         01        -- Subitem 3: CBOR short for Matcher Protection Type So
ftware
```

10.9. Biometric Authenticator Performance Bounds Extension (biometricPerfBounds)

This registration extension allows Relying Parties to specify the desired 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 authenticator allowed by the Relying Party.

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

Client extension processing
: 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 match the bounds as provided. The client can obtain information about the biometric authenticator's performance from authoritative sources such as the FIDO Metadata Service [FIDOMetadataService] (see Sec. 3.2 of [FIDOUAFAuthenticatorMetadataStatements]).

Client extension output
: Returns the JSON value true to indicate to the RP that the

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-Registries].

* 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 Format of this specification
* WebAuthn Attestation Statement Format Identifier: tpm
* Description: The TPM attestation statement format returns an attestation statement in the same format as the packed attestation statement format, although the 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-Registries].

* WebAuthn Extension Identifier: appid
* Description: This authentication extension allows Relying Parties that have previously registered a credential using the legacy FIDO JavaScript APIs to request an assertion.
* Specification Document: Section 10.1 FIDO AppId Extension (appid) of this specification
* WebAuthn Extension Identifier: txAuthSimple
* Description: This registration extension and authentication extension allows for a simple form of transaction authorization. A WebAuthn Relying Party can specify a prompt string, intended for display on a trusted device on the authenticator
* Specification Document: Section 10.2 Simple Transaction Authorization Extension (txAuthSimple) of this specification
* WebAuthn Extension Identifier: txAuthGeneric
* Description: This registration extension and authentication

extension was acted upon

Authenticator extension input
None.

Authenticator extension processing
None.

Authenticator extension output
None.

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-Registries].

* 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 Format of this specification
* WebAuthn Attestation Statement Format Identifier: tpm
* Description: The TPM attestation statement format returns an attestation statement in the same format as the packed attestation statement format, although the 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-Registries].

* WebAuthn Extension Identifier: appid
* Description: This authentication extension allows Relying Parties that have previously registered a credential using the legacy FIDO JavaScript APIs to request an assertion.
* Specification Document: Section 10.1 FIDO AppID Extension (appid) of this specification
* WebAuthn Extension Identifier: txAuthSimple
* Description: This registration extension and authentication extension allows for a simple form of transaction authorization. A WebAuthn Relying Party can specify a prompt string, intended for display on a trusted device on the authenticator
* Specification Document: Section 10.2 Simple Transaction Authorization Extension (txAuthSimple) of this specification
* WebAuthn Extension Identifier: txAuthGeneric
* Description: This registration extension and authentication

extension allows images to be used as transaction authorization prompts as well. This allows authenticators without a font rendering engine to be used and also supports a richer visual appearance than accomplished with the webauthn.txauth.simple extension.
* Specification Document: Section 10.3 Generic Transaction Authorization Extension (txAuthGeneric) of this specification
* WebAuthn Extension Identifier: authnSel
* Description: This registration extension allows a WebAuthn Relying Party to guide the selection of the authenticator that will be leveraged when creating the credential. It is intended primarily for WebAuthn Relying Parties that wish to tightly control the experience around credential creation.
* Specification Document: Section 10.4 Authenticator Selection Extension (authnSel) of this specification
* WebAuthn Extension Identifier: exts
* Description: This registration extension enables the Relying Party to determine which extensions the authenticator supports. The extension data is a list (CBOR array) of extension identifiers encoded as UTF-8 Strings. This extension is added automatically by the authenticator. This extension can be added to attestation statements.
* Specification Document: Section 10.5 Supported Extensions Extension (exts) of this specification
* WebAuthn Extension Identifier: uvi
* Description: This registration extension and authentication extension enables use of a user verification index. The user verification index is a value uniquely identifying a user verification data record. The UVI data can be used by servers to understand whether an authentication was authorized by the exact same biometric data as the initial key generation. This allows the detection and prevention of "friendly fraud".
* Specification Document: Section 10.6 User Verification Index Extension (uvi) of this specification
* WebAuthn Extension Identifier: loc
* Description: The location registration extension and authentication extension provides the client device's current location to the WebAuthn relying party, if supported by the client device and subject to user consent.
* Specification Document: Section 10.7 Location Extension (loc) of this specification
* WebAuthn Extension Identifier: uvm
* Description: This registration extension and authentication extension enables use of a user verification method. The user verification method extension returns to the Webauthn relying party which user verification methods (factors) were used for the WebAuthn operation.
* Specification Document: Section 10.8 User Verification Method Extension (uvm) of this specification

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 Algorithms registry [IANA-COSE-ALGS-REG]. It also registers identifiers for ECDAA algorithms.
* Name: RS256
* Value: -257
* Description: RSASSA-PKCS1-v1_5 w/ SHA-256
* Reference: Section 8.2 of [RFC8017]
* Recommended: No
* Name: RS384
* Value: -258
* Description: RSASSA-PKCS1-v1_5 w/ SHA-384
* Reference: Section 8.2 of [RFC8017]
* Recommended: No
* Name: RS512
* Value: -259
* Description: RSASSA-PKCS1-v1_5 w/ SHA-512
* Reference: Section 8.2 of [RFC8017]

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 than accomplished with the webauthn.txauth.simple extension.
* Specification Document: Section 10.3 Generic Transaction Authorization Extension (txAuthGeneric) of this specification
* WebAuthn Extension Identifier: authnSel
* Description: This registration extension allows a WebAuthn Relying Party to guide the selection of the authenticator that will be leveraged when creating the credential. It is intended primarily for WebAuthn Relying Parties that wish to tightly control the experience around credential creation.
* Specification Document: Section 10.4 Authenticator Selection Extension (authnSel) of this specification
* WebAuthn Extension Identifier: exts
* Description: This registration extension enables the Relying Party to determine which extensions the authenticator supports. The extension data is a list (CBOR array) of extension identifiers encoded as UTF-8 Strings. This extension is added automatically by the authenticator. This extension can be added to attestation statements.
* Specification Document: Section 10.5 Supported Extensions Extension (exts) of this specification
* WebAuthn Extension Identifier: uvi
* Description: This registration extension and authentication extension enables use of a user verification index. The user verification index is a value uniquely identifying a user verification data record. The UVI data can be used by servers to understand whether an authentication was authorized by the exact same biometric data as the initial key generation. This allows the detection and prevention of "friendly fraud".
* Specification Document: Section 10.6 User Verification Index Extension (uvi) of this specification
* WebAuthn Extension Identifier: loc
* Description: The location registration extension and authentication extension provides the client device's current location to the WebAuthn relying party, if supported by the client device and subject to user consent.
* Specification Document: Section 10.7 Location Extension (loc) of this specification
* WebAuthn Extension Identifier: uvm
* Description: This registration extension and authentication extension enables use of a user verification method. The user verification method extension returns to the Webauthn relying party which user verification methods (factors) were used for the WebAuthn operation.
* Specification Document: Section 10.8 User Verification Method Extension (uvm) of this specification

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 Algorithms registry [IANA-COSE-ALGS-REG]. It also registers identifiers for ECDAA algorithms.
* Name: RS256
* Value: TBD (requested assignment -257)
* Description: RSASSA-PKCS1-v1_5 w/ SHA-256
* Reference: Section 8.2 of [RFC8017]
* Recommended: No
* Name: RS384
* Value: TBD (requested assignment -258)
* Description: RSASSA-PKCS1-v1_5 w/ SHA-384
* Reference: Section 8.2 of [RFC8017]
* Recommended: No
* Name: RS512
* Value: TBD (requested assignment -259)
* Description: RSASSA-PKCS1-v1_5 w/ SHA-512
* Reference: Section 8.2 of [RFC8017]

```
        * Recommended: No
        * Name: ED256
        * Value: -260
        * Description: TPM_ECC_BN_P256 curve w/ SHA-256
        * Reference: Section 4.2 of [FIDOEcdaaAlgorithm]
        * Recommended: Yes
        * Name: ED512
        * Value: -261
        * Description: ECC_BN_ISOP512 curve w/ SHA-512
        * Reference: Section 4.2 of [FIDOEcdaaAlgorithm]
        * Recommended: Yes
        * Name: RS1
        * Value: -262
        * Description: RSASSA-PKCS1-v1_5 w/ SHA-1
        * Reference: Section 8.2 of [RFC8017]
        * Recommended: No

## 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.
 5. The authenticator shows appropriate UI for the user to select the
    authenticator on which the new credential will be created, and
    obtains a biometric or other authorization gesture from the user.
 6. The authenticator returns a response to the client platform, which
    in turn returns a response to the Relying Party script. If the user
    declined to select an authenticator or provide authorization, an
    appropriate error is returned.
 7. If a new credential was created,
      + The Relying Party script sends the newly generated credential
        public key to the server, along with additional information
        such as attestation regarding the provenance and
        characteristics of the authenticator.
      + The server stores the credential public key in its database
        and associates it with the user as well as with the
        characteristics of authentication indicated by attestation,
        also storing a friendly name for later use.
```

```
          + The script may store data such as the credential ID in local
            storage, to improve future UX by narrowing the choice of
            credential for the user.

   The sample code for generating and registering a new key follows:
if (!PublicKeyCredential) { /* Platform not capable. Handle error. */ }

var publicKey = {
  // The challenge must be produced by the server, see the Security Consideratio
ns
  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("MIIBkzCCATigAwIBAjCCAZMwggE4oAMCAQIwggGTMII
="), c=>c.charCodeAt(0)),
    name: "john.p.smith@example.com",
    displayName: "John P. Smith",
    icon: "https://pics.acme.com/00/p/aBjjjpqPb.png"
  },

  // This Relying Party will accept either an ES256 or RS256 credential, but
  // prefers an ES256 credential.
  pubKeyCredParams: [
    {
      type: "public-key",
      alg: -7 // "ES256" as registered in the IANA COSE Algorithms registry
    },
    {
      type: "public-key",
      alg: -257 // Value registered by this specification for "RS256"
    }
  ],

  timeout: 60000,  // 1 minute
  excludeCredentials: [], // No exclude list of PKCredDescriptors
  extensions: {"loc": true}  // Include location information
                             // in attestation
};

// 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.
  }).catch(function (err) {
    // No acceptable authenticator or user refused consent. Handle appropriately
.
  });

  12.2. Registration Specifically with User Verifying Platform Authenticator

   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
```

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

var publicKey = {
  // The challenge must be produced by the server, see the Security Consideratio
ns
  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("MIIBkzCCATigAwIBAjCCAZMwggE4oAMCAQIwggGTMII
="), c=>c.charCodeAt(0)),
    name: "alex.p.mueller@example.com",
    displayName: "Alex P. Mller",
    icon: "https://pics.example.com/00/p/aBjjjpqPb.png"
  },

  // This Relying Party will accept either an ES256 or RS256 credential, but
  // prefers an ES256 credential.
  pubKeyCredParams: [
    {
      type: "public-key",
      alg: -7 // "ES256" as registered in the IANA COSE Algorithms registry
    },
    {
      type: "public-key",
      alg: -257 // Value registered by this specification for "RS256"
    }
  ],

  timeout: 60000,  // 1 minute
  excludeCredentials: [], // No exclude list of PKCredDescriptors
  extensions: {"loc": true}  // Include location information
                             // in attestation
};

// 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.
  }).catch(function (err) {
    // No acceptable authenticator or user refused consent. Handle appropriately
.
  });

  12.2. Registration Specifically with User Verifying Platform Authenticator

   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
```

```
4841      successful credential creation, the RP script conveys the new
4842      credential to the server.
4843  if (!PublicKeyCredential) { /* Platform not capable of the API. Handle error. */
4844  }
4845
4846  PublicKeyCredential.isUserVerifyingPlatformAuthenticatorAvailable()
4847     .then(function (userIntent) {
4848
4849        // If the user has affirmed willingness to register with RP using an ava
4850  ilable platform authenticator
4851        if (userIntent) {
4852           var publicKeyOptions = { /* Public key credential creation options.
4853  */};
4854
4855           // Create and register credentials.
4856           return navigator.credentials.create({ "publicKey": publicKeyOptions
4857  });
4858        } else {
4859
4860           // Record that the user does not intend to use a platform authentica
4861  tor
4862           // and default the user to a password-based flow in the future.
4863        }
4864
4865     }).then(function (newCredentialInfo) {
4866        // Send new credential info to server for verification and registration.
4867     }).catch( function(err) {
4868        // Something went wrong. Handle appropriately.
4869     });
4870
4871   12.3. Authentication
4872
4873   This is the flow when a user with an already registered credential
4874   visits a website and wants to authenticate using the credential.
4875    1. The user visits example.com, which serves up a script.
4876    2. The script asks the client platform for an Authentication
4877       Assertion, providing as much information as possible to narrow the
4878       choice of acceptable credentials for the user. This may be obtained
4879       from the data that was stored locally after registration, or by
4880       other means such as prompting the user for a username.
4881    3. The Relying Party script runs one of the code snippets below.
4882    4. The client platform searches for and locates the authenticator.
4883    5. The client platform connects to the authenticator, performing any
4884       pairing actions if necessary.
4885    6. The authenticator presents the user with a notification that their
4886       attention is required. On opening the notification, the user is
4887       shown a friendly selection menu of acceptable credentials using the
4888       account information provided when creating the credentials, along
4889       with some information on the origin that is requesting these keys.
4890    7. The authenticator obtains a biometric or other authorization
4891       gesture from the user.
4892    8. The authenticator returns a response to the client platform, which
4893       in turn returns a response to the Relying Party script. If the user
4894       declined to select a credential or provide an authorization, an
4895       appropriate error is returned.
4896    9. If an assertion was successfully generated and returned,
4897          + The script sends the assertion to the server.
4898          + The server examines the assertion, extracts the credential ID,
4899            looks up the registered credential public key it is database,
4900            and verifies the assertion's authentication signature. If
4901            valid, it looks up the identity associated with the
4902            assertion's credential ID; that identity is now authenticated.
4903            If the credential ID is not recognized by the server (e.g., it
4904            has been deregistered due to inactivity) then the
4905            authentication has failed; each Relying Party will handle this
4906            in its own way.
4907          + The server now does whatever it would otherwise do upon
4908            successful authentication -- return a success page, set
4909            authentication cookies, etc.
```

```
5271      successful credential creation, the RP script conveys the new
5272      credential to the server.
5273
5274  if (!window.PublicKeyCredential) { /* Platform not capable of the API. Handle er
5275  ror. */ }
5276
5277  PublicKeyCredential.isUserVerifyingPlatformAuthenticatorAvailable()
5278     .then(function (userIntent) {
5279
5280        // If the user has affirmed willingness to register with RP using an ava
5281  ilable platform authenticator
5282        if (userIntent) {
5283           var publicKeyOptions = { /* Public key credential creation options.
5284  */};
5285
5286           // Create and register credentials.
5287           return navigator.credentials.create({ "publicKey": publicKeyOptions
5288  });
5289        } else {
5290
5291           // Record that the user does not intend to use a platform authentica
5292  tor
5293           // and default the user to a password-based flow in the future.
5294        }
5295
5296     }).then(function (newCredentialInfo) {
5297        // Send new credential info to server for verification and registration.
5298     }).catch( function(err) {
5299        // Something went wrong. Handle appropriately.
5300     });
5301
5302   12.3. Authentication
5303
5304   This is the flow when a user with an already registered credential
5305   visits a website and wants to authenticate using the credential.
5306    1. The user visits example.com, which serves up a script.
5307    2. The script asks the client platform for an Authentication
5308       Assertion, providing as much information as possible to narrow the
5309       choice of acceptable credentials for the user. This can be obtained
5310       from the data that was stored locally after registration, or by
5311       other means such as prompting the user for a username.
5312    3. The Relying Party script runs one of the code snippets below.
5313    4. The client platform searches for and locates the authenticator.
5314    5. The client platform connects to the authenticator, performing any
5315       pairing actions if necessary.
5316    6. The authenticator presents the user with a notification that their
5317       attention is needed. On opening the notification, the user is shown
5318       a friendly selection menu of acceptable credentials using the
5319       account information provided when creating the credentials, along
5320       with some information on the origin that is requesting these keys.
5321    7. The authenticator obtains a biometric or other authorization
5322       gesture from the user.
5323    8. The authenticator returns a response to the client platform, which
5324       in turn returns a response to the Relying Party script. If the user
5325       declined to select a credential or provide an authorization, an
5326       appropriate error is returned.
5327    9. If an assertion was successfully generated and returned,
5328          + The script sends the assertion to the server.
5329          + The server examines the assertion, extracts the credential ID,
5330            looks up the registered credential public key it is database,
5331            and verifies the assertion's authentication signature. If
5332            valid, it looks up the identity associated with the
5333            assertion's credential ID; that identity is now authenticated.
5334            If the credential ID is not recognized by the server (e.g., it
5335            has been deregistered due to inactivity) then the
5336            authentication has failed; each Relying Party will handle this
5337            in its own way.
5338          + The server now does whatever it would otherwise do upon
5339            successful authentication -- return a success page, set
5340            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:

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

var options = {
        // The challenge must be produced by the server, see the Security Considerations
        challenge: new Uint8Array([4,101,15 /* 29 more random bytes generated by the server */]),
        timeout: 60000,  // 1 minute
        allowCredentials: [{ type: "public-key" }]
      };

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

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

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

var options = {
        // The challenge must be produced by the server, see the Security Considerations
        challenge: new Uint8Array([8,18,33 /* 29 more random bytes generated by the server */]),
        timeout: 60000,  // 1 minute
        allowCredentials: [acceptableCredential1, acceptableCredential2],
        extensions: { 'txAuthSimple':
          "Wave your hands in the air like you just don't care" }
      };

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

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.

```
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.
```


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:

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

var options = {
        // The challenge must be produced by the server, see the Security Considerations
        challenge: new Uint8Array([4,101,15 /* 29 more random bytes generated by the server */]),
        timeout: 60000,  // 1 minute
        allowCredentials: [{ type: "public-key" }]
      };

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

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

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

var options = {
        // The challenge must be produced by the server, see the Security Considerations
        challenge: new Uint8Array([8,18,33 /* 29 more random bytes generated by the server */]),
        timeout: 60000,  // 1 minute
        allowCredentials: [acceptableCredential1, acceptableCredential2],
        extensions: { 'txAuthSimple':
          "Wave your hands in the air like you just don't care" }
      };

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

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.

```
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.
```

```
}

var options = {
   // A list of options.
}

navigator.credentials.create({
   publicKey: options,
   signal: authAbortSignal})
   .then(function (attestation) {
      // Register the user.
   }).catch(function (error) {
      if (error == "AbortError") {
         // Inform user the credential hasn't been created.
         // Let the server know a key hasn't been created.
      }
   });

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

13. Security Considerations


```
}

var options = {
   // A list of options.
}

navigator.credentials.create({
   publicKey: options,
   signal: authAbortSignal})
   .then(function (attestation) {
      // Register the user.
   }).catch(function (error) {
      if (error == "AbortError") {
         // Inform user the credential hasn't been created.
         // Let the server know a key hasn't been created.
      }
   });

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

13. Security Considerations

This specification defines a Web API and a cryptographic peer-entity authentication protocol. The Web Authentication API allows Web developers (i.e., "authors") to utilize the Web Authentication protocol in their registration and authentication ceremonies. The entities comprising the Web Authentication protocol endpoints are user-controlled authenticators and a Relying Party's computing environment hosting the Relying Party's web application. In this model, the user agent, together with the WebAuthn Client, comprise an intermediary between authenticators and Relying Parties. Additionally, authenticators can attest to Relying Parties as to their provenance.

At this time, this specification does not feature detailed security considerations. However, the [FIDOSecRef] document provides a security

13.1. Cryptographic Challenges

As a cryptographic protocol, Web Authentication is dependent upon randomized challenges to avoid replay attacks. Therefore, both {MakePublicKeyCredentialOptions/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.

14. Acknowledgements

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

analysis which is overall applicable to this specification. Also, the [FIDOAuthnrSecReqs] document suite defines authenticator security characteristics which are overall applicable for WebAuthn authenticators.

The below subsections comprise the current Web Authentication-specific security considerations.

13.1. 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 Relying Parties in an environment they trust (e.g., on the server-side), and the returned challenge value 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.

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 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 remember intermediate attestation CA certificates during Authenticator registration in order to un-register related public key credentials if the registration was performed after revocation of such certificates.

If an ECDAA attestation key has been compromised, it can be added to the RogueList (i.e., the list of revoked authenticators) maintained by the related ECDAA-Issuer. The Relying Party SHOULD verify whether an authenticator belongs to the RogueList when performing ECDAA-Verify (see section 3.6 in [FIDOEcdaaAlgorithm]). For example, the FIDO Metadata Service [FIDOMetadataService] provides one way to access such information.

13.3. credentialId Unsigned

The credential ID is not signed. This is not a problem because all that would happen if an authenticator returns the wrong credential ID, or if an attacker intercepts and manipulates the credential ID, is that the Relying Party would not look up the correct credential public key with which to verify the returned signed authenticator data (a.k.a., assertion), and thus the interaction would end in an error.

13.4. Browser Permissions Framework and Extensions

Web Authentication API implementations should leverage the browser permissions framework as much as possible when obtaining user permissions for certain extensions. An example is the location extension (see 10.7 Location Extension (loc)), implementations of which should make use of the existing browser permissions framework for the Geolocation API.

14. Privacy Considerations

The privacy principles in [FIDO-Privacy-Principles] also apply to this specification.

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.
  [UAFProtocol] 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.
  Note: In various places outside this specification, the term "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,

Boris Zbarsky.

Index

Terms defined by this specification

* aaguid, in 6.3.1
* AAGUID, in 10.4
* alg, in 5.3

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 not available.
* 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: Yuriy Ackermann, James Barclay, Richard Barnes, Dominic Battr, John Bradley, Domenic Denicola, Rahul Ghosh, Brad Hill, Jing Jin, Wally Jones, Ian Kilpatrick, Axel Nennker, Yoshikazu Nojima, Kimberly Paulhamus, Adam Powers, Yaron Sheffer, Anne van Kesteren, and Boris Zbarsky.

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.

Index

Terms defined by this specification

* AAGUID, in 10.4
* aaguid, in 6.3.1
* alg, in 5.3

```
5066 |     * allowCredentials, in 5.5
5067 |     * Assertion, in 4
5068 |     * assertion signature, in 6
5069 |     * attachment modality, in 5.4.5
5070 |     * Attestation, in 4
5071 |     * attestation, in 5.4
5072 |     * Attestation Certificate, in 4
5073 |     * Attestation Conveyance, in 5.4.6
5074 |     * AttestationConveyancePreference, in 5.4.6
5075 |     * attestationConveyancePreferenceOption, in 5.1.3
5076 |     * attestation key pair, in 4
5077 |     * attestationObject, in 5.2.1
5078 |     * attestation object, in 6.3
5079 |     * attestationObjectResult, in 5.1.3
5080 |     * attestation private key, in 4
5081 |     * attestation public key, in 4
5082 |     * attestation signature, in 6
5083 |     * attestation statement, in 6.3
5084 |     * attestation statement format, in 6.3
5085 |     * attestation statement format identifier, in 8.1
5086 |     * attestation trust path, in 6.3.2
5087 |     * attestation type, in 6.3
5088 |     * Attested credential data, in 6.3.1
5089 |     * attestedCredentialData, in 6.1
5090 |     * authDataExtensions, in 6.1
5091 |     * Authentication, in 4
5092 |     * Authentication Assertion, in 4
5093 |     * authentication extension, in 9
5094 |     * AuthenticationExtensions
5095 |         + definition of, in 5.7
5096 |         + (typedef), in 5.7
5097 |     * Authenticator, in 4
5098 |     * AuthenticatorAssertionResponse, in 5.2.2
5099 |     * AuthenticatorAttachment, in 5.4.5
5100 |     * authenticatorAttachment, in 5.4.4
5101 |     * AuthenticatorAttestationResponse, in 5.2.1
5102 |     * authenticatorCancel, in 6.2.3
5103 |     * authenticator data, in 6.1
5104 |     * authenticatorData, in 5.2.2
5105 |     * authenticator data claimed to have been used for the attestation,
5106 |       in 6.3.2
5107 |     * authenticator data for the attestation, in 6.3.2
5108 |     * authenticatorDataResult, in 5.1.4.1
5109 |     * authenticator extension, in 9
5110 |     * authenticator extension input, in 9.3
5111 |     * authenticator extension output, in 9.5
5112 |     * Authenticator extension processing, in 9.5
5113 |     * authenticatorExtensions, in 5.8.1
5114 |     * authenticatorGetAssertion, in 6.2.2
5115 |     * authenticatorMakeCredential, in 6.2.1
5116 |     * AuthenticatorResponse, in 5.2
5117 |     * authenticatorSelection, in 5.4
5118 |     * AuthenticatorSelectionCriteria, in 5.4.4
5119 |     * AuthenticatorSelectionList, in 10.4
```

```
5688 |     * allowCredentials, in 5.5
5689 |     * Anonymization CA, in 14.1
5690 |     * appid
5691 |         + dict-member for AuthenticationExtensionsClientInputs, in 10.1
5692 |         + dict-member for AuthenticationExtensionsClientOutputs, in
5693 |           10.1
5694 |     * Assertion, in 4
5695 |     * assertion signature, in 6
5696 |     * attachment modality, in 5.4.5
5697 |     * AttCA, in 6.3.3
5698 |     * Attestation, in 4
5699 |     * attestation, in 5.4
5700 |     * Attestation CA, in 6.3.3
5701 |     * Attestation Certificate, in 4
5702 |     * Attestation Conveyance, in 5.4.6
5703 |     * AttestationConveyancePreference, in 5.4.6
5704 |     * attestationConveyancePreferenceOption, in 5.1.3
5705 |     * attestation key pair, in 4
5706 |     * attestationObject, in 5.2.1
5707 |     * attestation object, in 6.3
5708 |     * attestationObjectResult, in 5.1.3
5709 |     * attestation private key, in 4
5710 |     * attestation public key, in 4
5711 |     * attestation signature, in 6
5712 |     * attestation statement, in 6.3
5713 |     * attestation statement format, in 6.3
5714 |     * attestation statement format identifier, in 8.1
5715 |     * attestation trust path, in 6.3.2
5716 |     * attestation type, in 6.3
5717 |     * Attested credential data, in 6.3.1
5718 |     * attestedCredentialData, in 6.1
5719 |     * authDataExtensions, in 6.1
5720 |     * Authentication, in 4
5721 |     * Authentication Assertion, in 4
5722 |     * authentication extension, in 9
5723 |     * AuthenticationExtensionsAuthenticatorInputs
5724 |         + definition of, in 5.9
5725 |         + (typedef), in 5.9
5726 |     * AuthenticationExtensionsClientInputs
5727 |         + definition of, in 5.7
5728 |         + (dictionary), in 5.7
5729 |     * AuthenticationExtensionsClientOutputs
5730 |         + definition of, in 5.8
5731 |         + (dictionary), in 5.8
5732 |     * AuthenticationExtensionsSupported, in 10.5
5733 |     * Authenticator, in 4
5734 |     * AuthenticatorAssertionResponse, in 5.2.2
5735 |     * AuthenticatorAttachment, in 5.4.5
5736 |     * authenticatorAttachment, in 5.4.4
5737 |     * AuthenticatorAttestationResponse, in 5.2.1
5738 |     * authenticatorBiometricPerfBounds, in 10.9
5739 |     * authenticatorCancel, in 6.2.4
5740 |     * authenticator data, in 6.1
5741 |     * authenticatorData, in 5.2.2
5742 |     * authenticator data claimed to have been used for the attestation,
5743 |       in 6.3.2
5744 |     * authenticator data for the attestation, in 6.3.2
5745 |     * authenticatorDataResult, in 5.1.4.1
5746 |     * authenticator extension, in 9
5747 |     * authenticator extension input, in 9.3
5748 |     * authenticator extension output, in 9.5
5749 |     * Authenticator extension processing, in 9.5
5750 |     * authenticatorGetAssertion, in 6.2.3
5751 |     * authenticatorMakeCredential, in 6.2.2
5752 |     * Authenticator Model, in 6
5753 |     * Authenticator operations, in 6.2
5754 |     * AuthenticatorResponse, in 5.2
5755 |     * authenticatorSelection, in 5.4
5756 |     * AuthenticatorSelectionCriteria, in 5.4.4
5757 |     * AuthenticatorSelectionList, in 10.4
```

```
* authenticator session, in 6.2
* AuthenticatorTransport, in 5.8.4
* Authorization Gesture, in 4
* Base64url Encoding, in 3
* Basic Attestation, in 6.3.3
* Biometric Recognition, in 4
* ble, in 5.8.4
* CBOR, in 3
* Ceremony, in 4
* challenge
    + dict-member for MakePublicKeyCredentialOptions, in 5.4
    + dict-member for PublicKeyCredentialRequestOptions, in 5.5
    + dict-member for CollectedClientData, in 5.8.1
* 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, in 9
* client extension input, in 9.3
* client extension output, in 9.4
* Client extension processing, in 9.4
* clientExtensionResults
    + dfn for credentialCreationData, in 5.1.3
    + dfn for assertionCreationData, in 5.1.4.1
* clientExtensions, in 5.8.1
* [[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 5.8.5
* [[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
* credentialPublicKey, in 6.3.1
* "cross-platform", in 5.4.5
* cross-platform, in 5.4.5
* cross-platform attached, in 5.4.5
* cross-platform attachment, in 5.4.5
* DAA, in 6.3.3
* direct, in 5.4.6
* "discouraged", in 5.8.6
* discouraged, in 5.8.6
* [[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
```

```
* authenticator session, in 6.2
* 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
* client data, in 5.10.1
* clientDataJSON, in 5.2
* clientDataJSONResult
    + dfn for credentialCreationData, in 5.1.3
    + dfn for assertionCreationData, in 5.1.4.1
* client extension, in 9
* client extension input, in 9.3
* client extension output, in 9.4
* Client extension processing, in 9.4
* clientExtensionResults
    + dfn for credentialCreationData, in 5.1.3
    + dfn for assertionCreationData, in 5.1.4.1
* [[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 5.10.1
* [[CollectFromCredentialStore]](origin, options,
  sameOriginWithAncestors), in 5.1.4
* Conforming User Agent, in 4
* content, in 10.3
* 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
* credentialPublicKey, in 6.3.1
* credentials map, in 6
* "cross-platform", in 5.4.5
* cross-platform, in 5.4.5
* cross-platform attached, in 5.4.5
* cross-platform attachment, in 5.4.5
* DAA, in 6.3.3
* direct, in 5.4.6
* "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 MakePublicKeyCredentialOptions, in 5.4
  + dict-member for PublicKeyCredentialRequestOptions, in 5.5
* flags, in 6.1
* getClientExtensionResults(), in 5.1
* hashAlgorithm, in 5.8.1
* 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
* identifier of the ECDAA-Issuer public key, in 8.2
* 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
* none, in 5.4.6
* 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 6.3.3
* pubKeyCredParams, in 5.4
* publicKey
  + dict-member for CredentialCreationOptions, in 5.1.1
  + dict-member for CredentialRequestOptions, in 5.1.2
* public-key, in 5.8.2
* Public Key Credential, in 4
* PublicKeyCredential, in 5.1
* PublicKeyCredentialDescriptor, in 5.8.3
* PublicKeyCredentialEntity, in 5.4.1
* PublicKeyCredentialParameters, in 5.3

* 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
* FRR, 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 PublicKeyCredentialRpEntity, 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
* identifier of the ECDAA-Issuer public key, in 8.2
* indirect, in 5.4.6
* isUserVerifyingPlatformAuthenticatorAvailable(), in 5.1.7
* 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 5.10.4
* No attestation statement, in 6.3.3
* None, in 6.3.3
* none, in 5.4.6
* none attestation statement format, in 8.7
* "not-supported", in 5.10.1
* not-supported, in 5.10.1
* origin, in 5.10.1
* otherUI, in 4
* platform, in 5.4.5
* "platform", in 5.4.5
* platform attachment, in 5.4.5
* platform authenticators, in 5.4.5
* platform credential, in 5.4.5
* "preferred", in 5.10.6
* preferred, in 5.10.6
* "present", in 5.10.1
* present, in 5.10.1
* [[preventSilentAccess]](credential, sameOriginWithAncestors), in 5.1.6
* privateKey, in 4
* pubKeyCredParams, in 5.4
* publicKey
  + dict-member for CredentialCreationOptions, in 5.1.1
  + dict-member for CredentialRequestOptions, in 5.1.2
* public-key, in 5.10.2
* Public Key Credential, in 4
* PublicKeyCredential, in 5.1
* PublicKeyCredentialCreationOptions, in 5.4
* PublicKeyCredentialDescriptor, in 5.10.3
* PublicKeyCredentialEntity, in 5.4.1
* PublicKeyCredentialParameters, in 5.3

* PublicKeyCredentialRequestOptions, in 5.5
* PublicKeyCredentialRpEntity, in 5.4.2
* Public Key Credential Source, in 4
* PublicKeyCredentialType, in 5.8.2
* PublicKeyCredentialUserEntity, in 5.4.3
* Rate Limiting, in 4
* rawId, in 5.1
* Registration, in 4
* registration extension, in 9
* Relying Party, in 4
* Relying Party Identifier, in 4
* "required", in 5.8.6
* required, in 5.8.6
* requireResidentKey, in 5.4.4
* response, in 5.1
* roaming authenticators, in 5.4.5
* rp, in 5.4
* rpId, in 5.5
* RP ID, in 4
* rpIdHash, in 6.1
* Self Attestation, in 6.3.3
* signature, in 5.2.2
* Signature Counter, in 6.1.1
* signatureResult, in 5.1.4.1
* signCount, in 6.1
* Signing procedure, in 6.3.2
* [[Store]](credential, sameOriginWithAncestors), in 5.1.5
* Test of User Presence, in 4
* timeout
    + dict-member for MakePublicKeyCredentialOptions, in 5.4
    + dict-member for PublicKeyCredentialRequestOptions, in 5.5
* tokenBindingId, in 5.8.1
* transports, in 5.8.3
* [[type]], in 5.1
* type
    + dict-member for PublicKeyCredentialParameters, in 5.3
    + dict-member for CollectedClientData, in 5.8.1
    + dict-member for PublicKeyCredentialDescriptor, in 5.8.3
* UP, in 4
* usb, in 5.8.4
* user, in 5.4
* User Consent, in 4
* userHandle, in 5.2.2
* User Handle, in 4
* userHandleResult, in 5.1.4.1
* User Present, in 4
* userVerification
    + dict-member for AuthenticatorSelectionCriteria, in 5.4.4
    + dict-member for PublicKeyCredentialRequestOptions, in 5.5

* PublicKeyCredentialRequestOptions, in 5.5
* PublicKeyCredentialRpEntity, in 5.4.2
* Public Key Credential Source, in 4
* PublicKeyCredentialType, in 5.10.2
* PublicKeyCredentialUserEntity, in 5.4.3
* Rate Limiting, in 4
* rawId, in 5.1
* Registration, in 4
* registration extension, in 9
* Relying Party, in 4
* Relying Party Identifier, in 4
* "required", in 5.10.6
* required, in 5.10.6
* requireResidentKey, in 5.4.4
* response, in 5.1
* roaming authenticators, in 5.4.5
* roaming credential, in 5.4.5
* rp, in 5.4
* rpId
    + dfn for public key credential source, in 4
    + dict-member for PublicKeyCredentialRequestOptions, in 5.5
* RP ID, in 4
* rpIdHash, in 6.1
* Self, in 6.3.3
* Self Attestation, in 6.3.3
* signature, in 5.2.2
* Signature Counter, in 6.1.1
* signatureResult, in 5.1.4.1
* signCount, in 6.1
* Signing procedure, in 6.3.2
* status, in 5.10.1
* [[Store]](credential, sameOriginWithAncestors), in 5.1.5
* supported, in 5.10.1
* "supported", in 5.10.1
* Test of User Presence, in 4
* timeout
    + dict-member for PublicKeyCredentialCreationOptions, in 5.4
    + dict-member for PublicKeyCredentialRequestOptions, in 5.5
* tokenBinding, in 5.10.1
* TokenBinding, in 5.10.1
* TokenBindingStatus, in 5.10.1
* transports, in 5.10.3
* txAuthGeneric
    + dict-member for AuthenticationExtensionsClientInputs, in 10.3
    + dict-member for AuthenticationExtensionsClientOutputs, in 10.3
* txAuthGenericArg, in 10.3
* txAuthSimple
    + dict-member for AuthenticationExtensionsClientInputs, in 10.2
    + dict-member for AuthenticationExtensionsClientOutputs, in 10.2
* [[type]], in 5.1
* type
    + dfn for public key credential source, in 4
    + dict-member for PublicKeyCredentialParameters, in 5.3
    + dict-member for CollectedClientData, in 5.10.1
    + dict-member for PublicKeyCredentialDescriptor, in 5.10.3
* UP, in 4
* usb, in 5.10.4
* user, in 5.4
* User Consent, in 4
* userHandle
    + dfn for public key credential source, in 4
    + attribute for AuthenticatorAssertionResponse, in 5.2.2
* User Handle, in 4
* userHandleResult, in 5.1.4.1
* User Present, in 4
* userVerification
    + dict-member for AuthenticatorSelectionCriteria, in 5.4.4
    + dict-member for PublicKeyCredentialRequestOptions, in 5.5

```
5276        * User Verification, in 4
5277        * UserVerificationRequirement, in 5.8.6
5278        * User Verified, in 4
5279        * UV, in 4
5280        * Verification procedure, in 6.3.2
5281        * verification procedure inputs, in 6.3.2
5282        * Web Authentication API, in 5
5283        * WebAuthn Client, in 4
5284
5285      Terms defined by reference
5286
5287        * [CREDENTIAL-MANAGEMENT-1] defines the following terms:
5288            + Credential
5289            + CredentialCreationOptions
5290            + CredentialRequestOptions
5291            + CredentialsContainer
5292            + Request a Credential
5293            + [[CollectFromCredentialStore]](origin, options,
5294              sameOriginWithAncestors)
5295            + [[Create]](origin, options, sameOriginWithAncestors)
5296            + [[Store]](credential, sameOriginWithAncestors)
5297            + [[discovery]]
5298            + [[type]]
5299            + create()
5300            + credential
5301            + credential source
5302            + get()
5303            + id
5304            + remote
5305            + same-origin with its ancestors
5306            + signal (for CredentialCreationOptions)
5307            + signal (for CredentialRequestOptions)
5308            + store()
5309            + type
5310            + user mediation
5311        * [DOM4] defines the following terms:
5312            + AbortController
5313            + aborted flag
5314            + document
5315        * [ECMAScript] defines the following terms:
5316            + %arraybuffer%
5317            + internal method
5318            + internal slot
5319            + stringify
5320        * [ENCODING] defines the following terms:
5321            + utf-8 encode
5322        * [FETCH] defines the following terms:
5323            + window
5324        * [HTML] defines the following terms:
5325            + ascii serialization of an origin
5326            + effective domain
5327            + environment settings object
```

```
5967        * User Verification, in 4
5968        * UserVerificationRequirement, in 5.10.6
5969        * User Verified, in 4
5970        * UV, in 4
5971        * uvi
5972            + dict-member for AuthenticationExtensionsClientInputs, in 10.6
5973            + dict-member for AuthenticationExtensionsClientOutputs, in
5974              10.6
5975        * uvm
5976            + dict-member for AuthenticationExtensionsClientInputs, in 10.8
5977            + dict-member for AuthenticationExtensionsClientOutputs, in
5978              10.8
5979        * UvmEntries, in 10.8
5980        * UvmEntry, in 10.8
5981        * Verification procedure, in 6.3.2
5982        * verification procedure inputs, in 6.3.2
5983        * Web Authentication API, in 5
5984        * WebAuthn Client, in 4
5985
5986      Terms defined by reference
5987
5988        * [CREDENTIAL-MANAGEMENT-1] defines the following terms:
5989            + Credential
5990            + CredentialCreationOptions
5991            + CredentialRequestOptions
5992            + CredentialsContainer
5993            + Request a Credential
5994            + [[CollectFromCredentialStore]](origin, options,
5995              sameOriginWithAncestors)
5996            + [[Create]](origin, options, sameOriginWithAncestors)
5997            + [[Store]](credential, sameOriginWithAncestors)
5998            + [[discovery]]
5999            + [[type]]
6000            + create()
6001            + credential
6002            + credential source
6003            + get()
6004            + id
6005            + remote
6006            + same-origin with its ancestors
6007            + signal (for CredentialCreationOptions)
6008            + signal (for CredentialRequestOptions)
6009            + store()
6010            + type
6011            + user mediation
6012        * [DOM4] defines the following terms:
6013            + AbortController
6014            + aborted flag
6015            + document
6016        * [ECMAScript] defines the following terms:
6017            + %arraybuffer%
6018            + internal method
6019            + internal slot
6020            + stringify
6021        * [ENCODING] defines the following terms:
6022            + utf-8 decode
6023            + utf-8 encode
6024        * [FETCH] defines the following terms:
6025            + window
6026        * [FIDO-APPID] defines the following terms:
6027            + determining if a caller's facetid is authorized for an appid
6028            + determining the facetid of a calling application
6029        * [FIDO-CTAP] defines the following terms:
6030            + ctap2 canonical cbor encoding form
6031        * [Geolocation-API] defines the following terms:
6032            + Coordinates
6033        * [HTML] defines the following terms:
6034            + ascii serialization of an origin
6035            + effective domain
6036            + environment settings object
```

```
5328          + global object
5329          + is a registrable domain suffix of or is equal to
5330          + is not a registrable domain suffix of and is not equal to
5331          + origin
5332          + relevant settings object
5333       * [HTML52] defines the following terms:
5334          + document.domain
5335          + opaque origin
5336          + origin
5337       * [INFRA] defines the following terms:
5338          + append (for list)
5339          + append (for set)
5340          + byte sequence
5341          + continue
5342          + empty
5343          + for each (for list)
5344          + for each (for map)
5345          + is empty
5346          + is not empty
5347          + item (for list)
5348          + item (for struct)
5349          + list
5350          + map
5351          + ordered set
5352          + remove
5353          + set
5354          + size
5355          + struct
5356          + while
5357          + willful violation
5358       * [mixed-content] defines the following terms:
5359          + a priori authenticated url
5360       * [page-visibility] defines the following terms:
5361          + visibility states
5362       * [secure-contexts] defines the following terms:
5363          + secure contexts
5364       * [TokenBinding] defines the following terms:
5365          + token binding
5366          + token binding id
5367       * [URL] defines the following terms:
5368          + domain
5369          + empty host
5370          + host
5371          + ipv4 address
5372          + ipv6 address
5373          + opaque host
5374          + url serializer
5375          + valid domain
5376          + valid domain string
5377       * [WebCryptoAPI] defines the following terms:
5378          + recognized algorithm name
5379       * [WebIDL] defines the following terms:
5380          + AbortError
5381          + ArrayBuffer
5382          + BufferSource
5383          + ConstraintError
5384          + DOMException
5385          + DOMString
5386          + Exposed
5387          + NotAllowedError
5388          + NotSupportedError
5389          + Promise
5390          + SameObject
5391          + SecureContext
5392          + SecurityError
5393          + USVString
5394          + UnknownError
5395          + boolean
```

```
6037          + global object
6038          + is a registrable domain suffix of or is equal to
6039          + is not a registrable domain suffix of and is not equal to
6040          + origin
6041          + relevant settings object
6042       * [HTML52] defines the following terms:
6043          + document.domain
6044          + opaque origin
6045          + origin
6046       * [INFRA] defines the following terms:
6047          + append (for list)
6048          + append (for set)
6049          + byte sequence
6050          + continue
6051          + for each (for list)
6052          + for each (for map)
6053          + is empty
6054          + is not empty
6055          + item (for list)
6056          + item (for struct)
6057          + list
6058          + map
6059          + ordered set
6060          + remove
6061          + set
6062          + set (for map)
6063          + struct
6064          + while
6065          + willful violation
6066       * [mixed-content] defines the following terms:
6067          + a priori authenticated url
6068       * [page-visibility] defines the following terms:
6069          + visibility states
6070       * [secure-contexts] defines the following terms:
6071          + secure contexts
6072       * [TokenBinding] defines the following terms:
6073          + token binding
6074          + token binding id
6075       * [URL] defines the following terms:
6076          + domain
6077          + empty host
6078          + host
6079          + ipv4 address
6080          + ipv6 address
6081          + opaque host
6082          + url serializer
6083          + valid domain
6084          + valid domain string
6085       * [WebIDL] defines the following terms:
6086          + AbortError
6087          + ArrayBuffer
6088          + BufferSource
6089          + ConstraintError
6090          + DOMException
6091          + DOMString
6092          + Exposed
6093          + InvalidStateError
6094          + NotAllowedError
6095          + NotSupportedError
6096          + Promise
6097          + SameObject
6098          + SecureContext
6099          + SecurityError
6100          + USVString
6101          + UnknownError
6102          + boolean
6103          + float
```

```
+ interface object
+ long
+ present
+ unsigned long
* [whatwg html] defines the following terms:
+ focus
```

```
+ interface object
+ long
+ present
+ unsigned long
* [whatwg html] defines the following terms:
+ focus
+ username
```

References

Normative References

[CDDL]
C. Vigano; H. Birkholz. CBOR data definition language (CDDL): a notational convention to express CBOR data structures. 21 September 2016. Internet Draft (work in progress). URL: https://tools.ietf.org/html/draft-greevenbosch-appsawg-cbor-cddl

[CREDENTIAL-MANAGEMENT-1]
Mike West. Credential Management Level 1. 4 August 2017. WD. URL: https://www.w3.org/TR/credential-management-1/

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

[ECMAScript]
ECMAScript Language Specification. URL: https://tc39.github.io/ecma262/

[ENCODING]
Anne van Kesteren. Encoding Standard. Living Standard. URL: https://encoding.spec.whatwg.org/

[FETCH]
Anne van Kesteren. Fetch Standard. Living Standard. URL: https://fetch.spec.whatwg.org/


```
[FIDO-APPID]
D. Balfanz; et al. FIDO AppID and Facets. FIDO Alliance Proposed Standard. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-appid-and-facets-v2.0-ps-20170927.html
```


[FIDO-CTAP]
R. Lindemann; et al. FIDO 2.0: Client to Authenticator Protocol. FIDO Alliance Review Draft. URL: https://fidoalliance.org/specs/fido-v2.0-rd-20170927/fido-client-to-authenticator-protocol-v2.0-rd-20170927.html


```
[FIDO-CTAP]
R. Lindemann; et al. FIDO 2.0: Client to Authenticator Protocol. FIDO Alliance Proposed Standard. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-client-to-authenticator-protocol-v2.0-ps-20170927.html
```

```
[FIDO-Privacy-Principles]
FIDO Alliance. FIDO Privacy Principles. FIDO Alliance Whitepaper. URL: https://fidoalliance.org/wp-content/uploads/2014/12/FIDO_Alliance_Whitepaper_Privacy_Principles.pdf
```


[FIDO-U2F-Message-Formats]
D. Balfanz; J. Ehrensvard; J. Lang. FIDO U2F Raw Message Formats. FIDO Alliance Implementation Draft. URL: https://fidoalliance.org/specs/fido-u2f-v1.1-id-20160915/fido-u2f-raw-message-formats-v1.1-id-20160915.html

[FIDOEcdaaAlgorithm]
R. Lindemann; et al. FIDO ECDAA Algorithm. FIDO Alliance Implementation Draft. URL: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-ecdaa-algorithm-v1.1-id-20170202.html

[FIDOReg]
R. Lindemann; D. 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]
Jatinder Mann; Arvind Jain. Page Visibility (Second Edition). 29 October 2013. REC. URL: https://www.w3.org/TR/page-visibility/

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

[RFC4648]
S. Josefsson. The Base16, Base32, and Base64 Data Encodings. October 2006. Proposed Standard. URL: https://tools.ietf.org/html/rfc4648

[RFC5234]
D. Crocker, Ed.; P. Overell. Augmented BNF for Syntax Specifications: ABNF. January 2008. Internet Standard. URL: https://tools.ietf.org/html/rfc5234

[RFC5890]
J. Klensin. Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework. August 2010. Proposed Standard. URL: https://tools.ietf.org/html/rfc5890

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

[RFC8152]
J. Schaad. CBOR Object Signing and Encryption (COSE). July 2017. Proposed Standard. URL: https://tools.ietf.org/html/rfc8152

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

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

[Geolocation-API]
Andrei Popescu. Geolocation API Specification 2nd Edition. 8 November 2016. REC. URL: https://www.w3.org/TR/geolocation-API/

[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. 14 December 2017. REC. 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]
Jatinder Mann; Arvind Jain. Page Visibility (Second Edition). 29 October 2013. REC. URL: https://www.w3.org/TR/page-visibility/

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

[RFC4648]
S. Josefsson. The Base16, Base32, and Base64 Data Encodings. October 2006. Proposed Standard. URL: https://tools.ietf.org/html/rfc4648

[RFC5234]
D. Crocker, Ed.; P. Overell. Augmented BNF for Syntax Specifications: ABNF. January 2008. Internet Standard. URL: https://tools.ietf.org/html/rfc5234

[RFC5890]
J. Klensin. Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework. August 2010. Proposed Standard. URL: https://tools.ietf.org/html/rfc5890

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

[RFC8152]
J. Schaad. CBOR Object Signing and Encryption (COSE). July 2017. Proposed Standard. URL: https://tools.ietf.org/html/rfc8152

[RFC8230]
M. Jones. Using RSA Algorithms with CBOR Object Signing and Encryption (COSE) Messages. September 2017. Proposed Standard. URL: https://tools.ietf.org/html/rfc8230

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

[SECURE-CONTEXTS]
Mike West. Secure Contexts. 15 September 2016. CR. URL: https://www.w3.org/TR/secure-contexts/

[TokenBinding]
A. Popov; et al. The Token Binding Protocol Version 1.0. February 16, 2017. Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-tokbind-protocol

[URL]
Anne van Kesteren. URL Standard. Living Standard. URL: https://url.spec.whatwg.org/

[WebAuthn-Registries]
Jeff Hodges; Giridhar Mandyam; Michael B. Jones. Registries for Web Authentication (WebAuthn). March 2017. Active Internet-Draft. URL: https://xml2rfc.tools.ietf.org/cgi-bin/xml2rfc.cgi?modeAsFormat=html/ascii&url=https://raw.githubusercontent.com/w3c/webauthn/master/draft-hodges-webauthn-registries.xml

[WebCryptoAPI]
Mark Watson. Web Cryptography API. 26 January 2017. REC. URL: https://www.w3.org/TR/WebCryptoAPI/

[WebIDL]
Cameron McCormack; Boris Zbarsky; Tobie Langel. Web IDL. 15 December 2016. ED. URL: https://heycam.github.io/webidl/

[WebIDL-1]
Cameron McCormack. WebIDL Level 1. 15 December 2016. REC. URL: https://www.w3.org/TR/2016/REC-WebIDL-1-20161215/

Informative References

[Ceremony]
Carl Ellison. Ceremony Design and Analysis. 2007. URL: https://eprint.iacr.org/2007/399.pdf

[Feature-Policy]
Feature Policy. Draft Community Group Report. URL: https://wicg.github.io/feature-policy/

[FIDO-APPID]
D. Balfanz; et al. FIDO AppID and Facets. FIDO Alliance Review Draft. URL: https://fidoalliance.org/specs/fido-uaf-v1.1-rd-20161005/fido-appid-and-facets-v1.1-rd-20161005.html

[FIDO-UAF-AUTHNR-CMDS]
R. Lindemann; J. Kemp. FIDO UAF Authenticator Commands. FIDO Alliance Implementation Draft. URL: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-authnr-cmds-v1.1-id-20170202.html

[SECURE-CONTEXTS]
Mike West. Secure Contexts. 15 September 2016. CR. URL: https://www.w3.org/TR/secure-contexts/

[TCG-CMCProfile-AIKCertEnroll]
Scott Kelly; et al. TCG Infrastructure Working Group: A CMC Profile for AIK Certificate Enrollment. 24 March 2011. Published. URL: https://trustedcomputinggroup.org/wp-content/uploads/IWG_CMC_Profile_Cert_Enrollment_v1_r7.pdf

[TokenBinding]
A. Popov; et al. The Token Binding Protocol Version 1.0. February 16, 2017. Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-tokbind-protocol

[URL]
Anne van Kesteren. URL Standard. Living Standard. URL: https://url.spec.whatwg.org/

[WebAuthn-Registries]
Jeff Hodges; Giridhar Mandyam; Michael B. Jones. Registries for Web Authentication (WebAuthn). March 2017. Active Internet-Draft. URL: https://xml2rfc.tools.ietf.org/cgi-bin/xml2rfc.cgi?modeAsFormat=html/ascii&url=https://raw.githubusercontent.com/w3c/webauthn/master/draft-hodges-webauthn-registries.xml

[WebIDL]
Cameron McCormack; Boris Zbarsky; Tobie Langel. Web IDL. 15 December 2016. ED. URL: https://heycam.github.io/webidl/

[WebIDL-1]
Cameron McCormack. WebIDL Level 1. 15 December 2016. REC. URL: https://www.w3.org/TR/2016/REC-WebIDL-1-20161215/

Informative References

[Ceremony]
Carl Ellison. Ceremony Design and Analysis. 2007. URL: https://eprint.iacr.org/2007/399.pdf

[EduPersonObjectClassSpec]
EduPerson Object Class Specification (200604a). May 15, 2007. URL: https://www.internet2.edu/media/medialibrary/2013/09/04/internet2-mace-dir-eduperson-200604.html

[Feature-Policy]
Feature Policy. Draft Community Group Report. URL: https://wicg.github.io/feature-policy/

[FIDO-UAF-AUTHNR-CMDS]
R. Lindemann; J. Kemp. FIDO UAF Authenticator Commands. FIDO Alliance Implementation Draft. URL: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-authnr-cmds-v1.1-id-20170202.html

[FIDOAuthnrSecReqs]

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

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

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

[ISOBiometricVocabulary]
ISO/IEC JTC1/SC37. Information technology -- Vocabulary -- Biometrics. 15 December 2012. International Standard: ISO/IEC 2382-37:2012(E) First Edition. URL: http://standards.iso.org/ittf/PubliclyAvailableStandards/c055194_ISOIEC_2382-37_2012.zip

[RFC4949]
R. Shirey. Internet Security Glossary, Version 2. August 2007. Informational. URL: https://tools.ietf.org/html/rfc4949

[RFC5280]
D. Cooper; et al. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. May 2008. Proposed Standard. URL: https://tools.ietf.org/html/rfc5280

[RFC6265]
A. Barth. HTTP State Management Mechanism. April 2011. Proposed Standard. URL: https://tools.ietf.org/html/rfc6265

[RFC6454]
A. Barth. The Web Origin Concept. December 2011. Proposed Standard. URL: https://tools.ietf.org/html/rfc6454

[RFC7515]
M. Jones; J. Bradley; N. Sakimura. JSON Web Signature (JWS). May 2015. Proposed Standard. URL: https://tools.ietf.org/html/rfc7515

[RFC8017]
K. Moriarty, Ed.; et al. PKCS #1: RSA Cryptography Specifications Version 2.2. November 2016. Informational. URL: https://tools.ietf.org/html/rfc8017

[TPMv2-EK-Profile]
TCG EK Credential Profile for TPM Family 2.0. URL: http://www.trustedcomputinggroup.org/wp-content/uploads/Credential_Profile_EK_V2.0_R14_published.pdf

[TPMv2-Part1]

D. Biggs; et al. FIDO Authenticator Security Requirements. FIDO Alliance Final Documents. URL: https://fidoalliance.org/specs/fido-security-requirements-v1.0-fd-20170524/

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

[FIDOSecRef]
R. Lindemann; D. Baghdasaryan; B. Hill. FIDO Security Reference. FIDO Alliance Proposed Standard. URL: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-security-ref-v1.2-ps-20170411.html

[FIDOUAFAuthenticatorMetadataStatements]
B. Hill; D. Baghdasaryan; J. Kemp. FIDO UAF Authenticator Metadata Statements v1.0. FIDO Alliance Proposed Standard. URL: https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-authnr-metadata-v1.0-ps-20141208.html

[ISOBiometricVocabulary]
ISO/IEC JTC1/SC37. Information technology -- Vocabulary -- Biometrics. 15 December 2012. International Standard: ISO/IEC 2382-37:2012(E) First Edition. URL: http://standards.iso.org/ittf/PubliclyAvailableStandards/c055194_ISOIEC_2382-37_2012.zip

[RFC3279]
L. Bassham; W. Polk; R. Housley. Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. April 2002. Proposed Standard. URL: https://tools.ietf.org/html/rfc3279

[RFC4949]
R. Shirey. Internet Security Glossary, Version 2. August 2007. Informational. URL: https://tools.ietf.org/html/rfc4949

[RFC5280]
D. Cooper; et al. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. May 2008. Proposed Standard. URL: https://tools.ietf.org/html/rfc5280

[RFC6265]
A. Barth. HTTP State Management Mechanism. April 2011. Proposed Standard. URL: https://tools.ietf.org/html/rfc6265

[RFC6454]
A. Barth. The Web Origin Concept. December 2011. Proposed Standard. URL: https://tools.ietf.org/html/rfc6454

[RFC7515]
M. Jones; J. Bradley; N. Sakimura. JSON Web Signature (JWS). May 2015. Proposed Standard. URL: https://tools.ietf.org/html/rfc7515

[RFC8017]
K. Moriarty, Ed.; et al. PKCS #1: RSA Cryptography Specifications Version 2.2. November 2016. Informational. URL: https://tools.ietf.org/html/rfc8017

[TPMv2-EK-Profile]
TCG EK Credential Profile for TPM Family 2.0. URL: http://www.trustedcomputinggroup.org/wp-content/uploads/Credential_Profile_EK_V2.0_R14_published.pdf

[TPMv2-Part1]

```
5627          Trusted Platform Module Library, Part 1: Architecture. URL:
5628          http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-
5629          2.0-Part-1-Architecture-01.38.pdf
5630
5631      [TPMv2-Part2]
5632          Trusted Platform Module Library, Part 2: Structures. URL:
5633          http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-
5634          2.0-Part-2-Structures-01.38.pdf
5635
5636      [TPMv2-Part3]
5637          Trusted Platform Module Library, Part 3: Commands. URL:
5638          http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-
5639          2.0-Part-3-Commands-01.38.pdf
5640
5641      [UAFProtocol]
5642          R. Lindemann; et al. FIDO UAF Protocol Specification v1.0. FIDO
5643          Alliance Proposed Standard. URL:
5644          https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-ua
5645          f-protocol-v1.0-ps-20141208.html
5646
5647  IDL Index
5648
5649  [SecureContext, Exposed=Window]
5650  interface PublicKeyCredential : Credential {
5651      [SameObject] readonly attribute ArrayBuffer             rawId;
5652      [SameObject] readonly attribute AuthenticatorResponse    response;
5653      AuthenticationExtensions getClientExtensionResults();
5654  };
5655
5656  partial dictionary CredentialCreationOptions {
5657      MakePublicKeyCredentialOptions      publicKey;
5658  };
5659
5660  partial dictionary CredentialRequestOptions {
5661      PublicKeyCredentialRequestOptions      publicKey;
5662  };
5663
5664  partial interface PublicKeyCredential {
5665      static Promise < boolean > isUserVerifyingPlatformAuthenticatorAvailable();
5666  };
5667
5668  [SecureContext, Exposed=Window]
5669  interface AuthenticatorResponse {
5670      [SameObject] readonly attribute ArrayBuffer      clientDataJSON;
5671  };
5672
5673  [SecureContext, Exposed=Window]
5674  interface AuthenticatorAttestationResponse : AuthenticatorResponse {
5675      [SameObject] readonly attribute ArrayBuffer      attestationObject;
5676  };
5677
5678  [SecureContext, Exposed=Window]
5679  interface AuthenticatorAssertionResponse : AuthenticatorResponse {
5680      [SameObject] readonly attribute ArrayBuffer      authenticatorData;
5681      [SameObject] readonly attribute ArrayBuffer      signature;
5682      [SameObject] readonly attribute ArrayBuffer     userHandle;
5683  };
5684
5685  dictionary PublicKeyCredentialParameters {
5686      required PublicKeyCredentialType      type;
5687      required COSEAlgorithmIdentifier      alg;
5688  };
5689
5690  dictionary MakePublicKeyCredentialOptions {
5691      required PublicKeyCredentialRpEntity         rp;
5692      required PublicKeyCredentialUserEntity       user;
5693
5694      required BufferSource                        challenge;
5695      required sequence<PublicKeyCredentialParameters>  pubKeyCredParams;
5696
```



```
6374          Trusted Platform Module Library, Part 1: Architecture. URL:
6375          http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-
6376          2.0-Part-1-Architecture-01.38.pdf
6377
6378      [TPMv2-Part2]
6379          Trusted Platform Module Library, Part 2: Structures. URL:
6380          http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-
6381          2.0-Part-2-Structures-01.38.pdf
6382
6383      [TPMv2-Part3]
6384          Trusted Platform Module Library, Part 3: Commands. URL:
6385          http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-
6386          2.0-Part-3-Commands-01.38.pdf
6387
6388      [UAFProtocol]
6389          R. Lindemann; et al. FIDO UAF Protocol Specification v1.0. FIDO
6390          Alliance Proposed Standard. URL:
6391          https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-ua
6392          f-protocol-v1.0-ps-20141208.html
6393
6394  IDL Index
6395
6396  [SecureContext, Exposed=Window]
6397  interface PublicKeyCredential : Credential {
6398      [SameObject] readonly attribute ArrayBuffer             rawId;
6399      [SameObject] readonly attribute AuthenticatorResponse    response;
6400      AuthenticationExtensionsClientOutputs getClientExtensionResults();
6401  };
6402
6403  partial dictionary CredentialCreationOptions {
6404      PublicKeyCredentialCreationOptions      publicKey;
6405  };
6406
6407  partial dictionary CredentialRequestOptions {
6408      PublicKeyCredentialRequestOptions      publicKey;
6409  };
6410
6411  partial interface PublicKeyCredential {
6412      static Promise < boolean > isUserVerifyingPlatformAuthenticatorAvailable();
6413  };
6414
6415  [SecureContext, Exposed=Window]
6416  interface AuthenticatorResponse {
6417      [SameObject] readonly attribute ArrayBuffer      clientDataJSON;
6418  };
6419
6420  [SecureContext, Exposed=Window]
6421  interface AuthenticatorAttestationResponse : AuthenticatorResponse {
6422      [SameObject] readonly attribute ArrayBuffer      attestationObject;
6423  };
6424
6425  [SecureContext, Exposed=Window]
6426  interface AuthenticatorAssertionResponse : AuthenticatorResponse {
6427      [SameObject] readonly attribute ArrayBuffer      authenticatorData;
6428      [SameObject] readonly attribute ArrayBuffer      signature;
6429      [SameObject] readonly attribute ArrayBuffer?     userHandle;
6430  };
6431
6432  dictionary PublicKeyCredentialParameters {
6433      required PublicKeyCredentialType      type;
6434      required COSEAlgorithmIdentifier      alg;
6435  };
6436
6437  dictionary PublicKeyCredentialCreationOptions {
6438      required PublicKeyCredentialRpEntity         rp;
6439      required PublicKeyCredentialUserEntity       user;
6440
6441      required BufferSource                        challenge;
6442      required sequence<PublicKeyCredentialParameters>  pubKeyCredParams;
6443
```

```
    unsigned long                       timeout;
    sequence<PublicKeyCredentialDescriptor>     excludeCredentials = [];
    AuthenticatorSelectionCriteria              authenticatorSelection;
    AttestationConveyancePreference             attestation = "none";
    AuthenticationExtensions                    extensions;
};

dictionary PublicKeyCredentialEntity {
    required DOMString    name;
    USVString             icon;
};

dictionary PublicKeyCredentialRpEntity : PublicKeyCredentialEntity {
    DOMString     id;
};

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

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

enum AuthenticatorAttachment {
    "platform",      // Platform attachment
    "cross-platform"  // Cross-platform attachment
};

enum AttestationConveyancePreference {
    "none",
    "indirect",
    "direct"
};

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

typedef record<DOMString, any>      AuthenticationExtensions;

dictionary CollectedClientData {
    required DOMString          type;
    required DOMString          challenge;
    required DOMString          origin;
    required DOMString          hashAlgorithm;
    DOMString                   tokenBindingId;
    AuthenticationExtensions    clientExtensions;
    AuthenticationExtensions    authenticatorExtensions;

};
```



```
    unsigned long                       timeout;
    sequence<PublicKeyCredentialDescriptor>     excludeCredentials = [];
    AuthenticatorSelectionCriteria              authenticatorSelection;
    AttestationConveyancePreference             attestation = "none";
    AuthenticationExtensionsClientInputs        extensions;
};

dictionary PublicKeyCredentialEntity {
    required DOMString    name;
    USVString             icon;
};

dictionary PublicKeyCredentialRpEntity : PublicKeyCredentialEntity {
    DOMString     id;
};

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

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

enum AuthenticatorAttachment {
    "platform",      // Platform attachment
    "cross-platform"  // Cross-platform attachment
};

enum AttestationConveyancePreference {
    "none",
    "indirect",
    "direct"
};

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

dictionary AuthenticationExtensionsClientInputs {
};

dictionary AuthenticationExtensionsClientOutputs {
};

typedef record<DOMString, DOMString> AuthenticationExtensionsAuthenticatorInputs
;

dictionary CollectedClientData {
    required DOMString          type;
    required DOMString          challenge;
    required DOMString          origin;
    TokenBinding                tokenBinding;
};

dictionary TokenBinding {
    required TokenBindingStatus status;
    DOMString id;
};

enum TokenBindingStatus { "present", "supported", "not-supported" };
```

```
    enum PublicKeyCredentialType {
        "public-key"
    };

    dictionary PublicKeyCredentialDescriptor {
        required PublicKeyCredentialType      type;
        required BufferSource                 id;
        sequence<AuthenticatorTransport>      transports;
    };

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

    typedef long COSEAlgorithmIdentifier;

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

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

```
    typedef BufferSource      AAGUID;
```



```
    enum PublicKeyCredentialType {
        "public-key"
    };

    dictionary PublicKeyCredentialDescriptor {
        required PublicKeyCredentialType      type;
        required BufferSource                 id;
        sequence<AuthenticatorTransport>      transports;
    };

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

    typedef long COSEAlgorithmIdentifier;

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

    partial dictionary AuthenticationExtensionsClientInputs {
      USVString appid;
    };

    partial dictionary AuthenticationExtensionsClientOutputs {
      boolean appid;
    };

    partial dictionary AuthenticationExtensionsClientInputs {
      USVString txAuthSimple;
    };

    partial dictionary AuthenticationExtensionsClientOutputs {
      USVString txAuthSimple;
    };

    dictionary txAuthGenericArg {
        required USVString contentType;    // MIME-Type of the content, e.g., "image
    /png"
        required ArrayBuffer content;
    };

    partial dictionary AuthenticationExtensionsClientInputs {
      txAuthGenericArg txAuthGeneric;
    };

    partial dictionary AuthenticationExtensionsClientOutputs {
      ArrayBuffer txAuthGeneric;
    };

    typedef sequence<AAGUID> AuthenticatorSelectionList;

    partial dictionary AuthenticationExtensionsClientInputs {
      AuthenticatorSelectionList authnSel;
    };

    typedef BufferSource      AAGUID;

    partial dictionary AuthenticationExtensionsClientOutputs {
      boolean authnSel;
    };

    partial dictionary AuthenticationExtensionsClientInputs {
      boolean exts;
    };
```

```
5784
5785  Issues Index
5786
5787    The definitions of "lifetime of" and "becomes available" are intended
5788    to represent how devices are hotplugged into (USB) or discovered by
5789    (NFC) browsers, and are under-specified. Resolving this with good
5790    definitions or some other means will be addressed by resolving Issue
5791    #613. RET
5792    need to define "blinding". See also #462.
5793    <https://github.com/w3c/webauthn/issues/694> RET
5794    @balfanz wishes to add to the "direct" case: If the authenticator
5795    violates the privacy requirements of the attestation type it is using,
5796    the client SHOULD terminate this algorithm with a
5797    "AttestationNotPrivateError". RET
5798    The definitions of "lifetime of" and "becomes available" are intended
5799    to represent how devices are hotplugged into (USB) or discovered by
5800    (NFC) browsers, and are under-specified. 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 global in which to allocate a place for it. Perhaps this is good
5806    enough? addendum: @jcjones 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    browsing 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
```

```
6584    typedef sequence<USVString> AuthenticationExtensionsSupported;
6585
6586    partial dictionary AuthenticationExtensionsClientOutputs {
6587      AuthenticationExtensionsSupported exts;
6588    };
6589
6590    partial dictionary AuthenticationExtensionsClientInputs {
6591      boolean uvi;
6592    };
6593
6594    partial dictionary AuthenticationExtensionsClientOutputs {
6595      ArrayBuffer uvi;
6596    };
6597
6598    partial dictionary AuthenticationExtensionsClientInputs {
6599      boolean loc;
6600    };
6601
6602    partial dictionary AuthenticationExtensionsClientOutputs {
6603      Coordinates loc;
6604    };
6605
6606    partial dictionary AuthenticationExtensionsClientInputs {
6607      boolean uvm;
6608    };
6609
6610    typedef sequence<unsigned long> UvmEntry;
6611    typedef sequence<UvmEntry> UvmEntries;
6612
6613    partial dictionary AuthenticationExtensionsClientOutputs {
6614      UvmEntries uvm;
6615    };
6616
6617    dictionary authenticatorBiometricPerfBounds{
6618       float FAR;
6619       float FRR;
6620       };
6621
6622
6623    Issues Index
6624
6625      The definitions of "lifetime of" and "becomes available" are intended
6626      to represent how devices are hot-plugged into (USB) or discovered by
6627      (NFC) browsers, and are underspecified. Resolving this with good
6628      definitions or some other means will be addressed by resolving Issue
6629      #613. RET

6630      @balfanz wishes to add to the "direct" case: If the authenticator
6631      violates the privacy requirements of the attestation type it is using,
6632      the client SHOULD terminate this algorithm with an
6633      "AttestationNotPrivateError". RET
6634      The definitions of "lifetime of" and "becomes available" are intended
6635      to represent how devices are hot-plugged into (USB) or discovered by
6636      (NFC) browsers, and are underspecified. Resolving this with good
6637      definitions or some other means will be addressed by resolving Issue
6638      #613. RET
6639      The foregoing step _may_ be incorrect, in that we are attempting to
6640      create savedCredentialId here and use it later below, and we do not
6641      have a global in which to allocate a place for it. Perhaps this is good
6642      enough? addendum: @jcjones feels the above step is likely good enough.
6643      RET
6644      The WHATWG HTML WG is discussing whether to provide a hook when a
6645      browsing context gains or loses focuses. If a hook is provided, the
6646      above paragraph will be updated to include the hook. See WHATWG HTML WG
6647      Issue #2711 for more details. RET
6648
6649      #base64url-encodingReferenced in:
6650        * 5.1. PublicKeyCredential Interface
6651        * 5.1.3. Create a new credential - PublicKeyCredential's
```

```
5816        [[Create]](origin, options, sameOriginWithAncestors) method (2)
5817      * 5.1.4.1. PublicKeyCredential's
5818        [[DiscoverFromExternalSource]](origin, options,
5819        sameOriginWithAncestors) method (2)
5820      * 7.2. Verifying an authentication assertion
5821
5822    #cborReferenced in:
5823      * 5.1.3. Create a new credential - PublicKeyCredential's
5824        [[Create]](origin, options, sameOriginWithAncestors) method
5825      * 5.1.4.1. PublicKeyCredential's
5826        [[DiscoverFromExternalSource]](origin, options,
5827        sameOriginWithAncestors) method
5828      * 6.1. Authenticator data (2)
5829      * 9. WebAuthn Extensions (2) (3)
5830      * 9.2. Defining extensions (2)
5831      * 9.3. Extending request parameters
5832      * 9.4. Client extension processing (2)
5833      * 9.5. Authenticator extension processing (2) (3) (4) (5)
5834
5835    #attestationReferenced in:
5836      * 4. Terminology (2)
5837      * 5.4.6. Attestation Conveyance Preference enumeration (enum
5838        AttestationConveyancePreference) (2)
5839      * 6. WebAuthn Authenticator model (2)
5840      * 6.3. Attestation (2) (3) (4)
5841      * 11.1. WebAuthn Attestation Statement Format Identifier
5842        Registrations
5843
5844    #attestation-certificateReferenced in:
5845      * 4. Terminology (2)
5846      * 5.1.3. Create a new credential - PublicKeyCredential's
5847        [[Create]](origin, options, sameOriginWithAncestors) method
5848      * 8.3.1. TPM attestation statement certificate requirements
5849
5850    #attestation-key-pairReferenced in:
5851      * 4. Terminology (2)
5852      * 6.3. Attestation
5853
5854    #attestation-private-keyReferenced in:
5855      * 6. WebAuthn Authenticator model
5856      * 6.3. Attestation
5857
5858    #attestation-public-keyReferenced in:
5859      * 6.3. Attestation
5860
5861    #authenticationReferenced in:
5862      * 1. Introduction (2)
5863      * 4. Terminology (2) (3) (4) (5) (6) (7)
5864      * 7.2. Verifying an authentication assertion (2) (3)
5865
5866    #authentication-assertionReferenced in:
```

```
6652        [[Create]](origin, options, sameOriginWithAncestors) method (2)
6653      * 5.1.4.1. PublicKeyCredential's
6654        [[DiscoverFromExternalSource]](origin, options,
6655        sameOriginWithAncestors) method (2)
6656      * 5.10.1. Client data used in WebAuthn signatures (dictionary
6657        CollectedClientData)
6658      * 7.1. Registering a new credential
6659      * 7.2. Verifying an authentication assertion (2)
6660
6661    #cborReferenced in:
6662      * 2.4. All Conformance Classes
6663      * 3. Dependencies
6664      * 5.1.3. Create a new credential - PublicKeyCredential's
6665        [[Create]](origin, options, sameOriginWithAncestors) method (2)
6666      * 5.1.4.1. PublicKeyCredential's
6667        [[DiscoverFromExternalSource]](origin, options,
6668        sameOriginWithAncestors) method
6669      * 6.1. Authenticator data (2)
6670      * 6.2.2. The authenticatorMakeCredential operation
6671      * 6.2.3. The authenticatorGetAssertion operation
6672      * 9. WebAuthn Extensions (2) (3) (4) (5) (6) (7)
6673      * 9.2. Defining extensions (2)
6674      * 9.3. Extending request parameters
6675      * 9.4. Client extension processing (2)
6676      * 9.5. Authenticator extension processing (2)
6677
6678    #assertionReferenced in:
6679      * 7.1. Registering a new credential
6680      * 10.1. FIDO AppID Extension (appid)
6681      * 13.3. credentialId Unsigned
6682
6683    #attestationReferenced in:
6684      * 4. Terminology (2)
6685      * 5.4.6. Attestation Conveyance Preference enumeration (enum
6686        AttestationConveyancePreference) (2)
6687      * 6. WebAuthn Authenticator Model (2)
6688      * 6.3. Attestation (2) (3) (4)
6689      * 8.2. Packed Attestation Statement Format
6690      * 11.1. WebAuthn Attestation Statement Format Identifier
6691        Registrations
6692      * 13. Security Considerations
6693
6694    #attestation-certificateReferenced in:
6695      * 4. Terminology (2)
6696      * 6.3.3. Attestation Types
6697      * 8.3.1. TPM attestation statement certificate requirements
6698
6699    #attestation-key-pairReferenced in:
6700      * 4. Terminology (2)
6701      * 6.3. Attestation
6702      * 6.3.3. Attestation Types
6703
6704    #attestation-private-keyReferenced in:
6705      * 6. WebAuthn Authenticator Model
6706      * 6.3. Attestation
6707      * 8.2. Packed Attestation Statement Format
6708
6709    #attestation-public-keyReferenced in:
6710      * 6.3. Attestation
6711      * 8.2. Packed Attestation Statement Format
6712
6713    #authenticationReferenced in:
6714      * 1. Introduction (2)
6715      * 4. Terminology (2) (3) (4) (5) (6) (7)
6716      * 7.2. Verifying an authentication assertion (2) (3) (4)
6717      * 13. Security Considerations
6718      * 14.3. Authentication Ceremony Privacy
6719
6720    #authentication-assertionReferenced in:
```

```
5926        * 1.1.2. Authentication
5927        * 1.1.3. Other use cases and configurations
5928        * 4. Terminology (2) (3) (4) (5) (6)
5929        * 5.1.4. Use an existing credential to make an assertion -
5930          PublicKeyCredential's [[Get]](options) method (2)
5931
5932      #biometric-recognitionReferenced in:
5933        * 4. Terminology (2)
5934
5935      #ceremonyReferenced in:
5936        * 1. Introduction
5937        * 4. Terminology (2) (3) (4) (5) (6) (7)
5938        * 7.1. Registering a new credential
5939        * 7.2. Verifying an authentication assertion
5940
5941      #clientReferenced in:
5942        * 4. Terminology
5943        * 5.1.6. Availability of User-Verifying Platform Authenticator -
5944          PublicKeyCredential's
5945          isUserVerifyingPlatformAuthenticatorAvailable() method (2) (3) (4)
5946
5947      #client-side-resident-credential-private-keyReferenced in:
5948        * 4. Terminology (2)
5949        * 5.1.3. Create a new credential - PublicKeyCredential's
5950          [[Create]](origin, options, sameOriginWithAncestors) method
5951        * 5.4.4. Authenticator Selection Criteria (dictionary
5952          AuthenticatorSelectionCriteria) (2)
5953        * 6.2.1. The authenticatorMakeCredential operation
5954
5955      #conforming-user-agentReferenced in:
5956        * 1. Introduction
5957        * 2.1. User Agents
5958        * 2.2. Authenticators
5959        * 4. Terminology (2)
5960
5961      #credential-idReferenced in:
5962        * 4. Terminology (2) (3) (4)
5963        * 5.1.4.1. PublicKeyCredential's
5964          [[DiscoverFromExternalSource]](origin, options,
5965          sameOriginWithAncestors) method
5966        * 5.2.1. Information about Public Key Credential (interface
5967          AuthenticatorAttestationResponse)
5968        * 6.2.2. The authenticatorGetAssertion operation (2)
5969        * 6.3.1. Attested credential data
5970        * 7.1. Registering a new credential
5971        * 8.6. FIDO U2F Attestation Statement Format
5972        * 12.1. Registration
5973        * 12.3. Authentication (2) (3)
5974
5975      #credential-public-keyReferenced in:
```

```
6788        * 1.1.2. Authentication
6789        * 1.1.3. Other use cases and configurations
6790        * 4. Terminology (2) (3) (4) (5) (6)
6791        * 5.1.4. Use an existing credential to make an assertion -
6792          PublicKeyCredential's [[Get]](options) method (2)
6793        * 5.1.6. Preventing silent access to an existing credential -
6794          PublicKeyCredential's [[preventSilentAccess]](credential,
6795          sameOriginWithAncestors) method
6796
6797      #biometric-recognitionReferenced in:
6798        * 4. Terminology (2) (3)
6799
6800      #biometric-authenticatorReferenced in:
6801        * 10.9. Biometric Authenticator Performance Bounds Extension
6802          (biometricPerfBounds)
6803
6804      #ceremonyReferenced in:
6805        * 1. Introduction
6806        * 4. Terminology (2) (3) (4) (5) (6) (7)
6807        * 7.1. Registering a new credential (2)
6808        * 7.2. Verifying an authentication assertion (2)
6809        * 13. Security Considerations
6810        * 14.2. Registration Ceremony Privacy
6811        * 14.3. Authentication Ceremony Privacy (2)
6812
6813      #clientReferenced in:
6814        * 4. Terminology
6815        * 5.1.7. Availability of User-Verifying Platform Authenticator -
6816          PublicKeyCredential's
6817          isUserVerifyingPlatformAuthenticatorAvailable() method (2) (3) (4)
6818        * 5.4.5. Authenticator Attachment enumeration (enum
6819          AuthenticatorAttachment) (2) (3)
6820        * 7.1. Registering a new credential
6821        * 7.2. Verifying an authentication assertion
6822
6823      #client-side-resident-credential-private-keyReferenced in:
6824        * 4. Terminology (2)
6825        * 5.1.3. Create a new credential - PublicKeyCredential's
6826          [[Create]](origin, options, sameOriginWithAncestors) method
6827        * 5.4.4. Authenticator Selection Criteria (dictionary
6828          AuthenticatorSelectionCriteria) (2)
6829        * 6.2.2. The authenticatorMakeCredential operation (2)
6830
6831      #conforming-user-agentReferenced in:
6832        * 1. Introduction
6833        * 2.1. User Agents
6834        * 2.2. Authenticators
6835        * 4. Terminology (2)
6836
6837      #credential-idReferenced in:
6838        * 4. Terminology (2) (3) (4)
6839        * 5.1. PublicKeyCredential Interface (2)
6840        * 5.1.4.1. PublicKeyCredential's
6841          [[DiscoverFromExternalSource]](origin, options,
6842          sameOriginWithAncestors) method
6843        * 5.2.1. Information about Public Key Credential (interface
6844          AuthenticatorAttestationResponse)
6845        * 5.10.3. Credential Descriptor (dictionary
6846          PublicKeyCredentialDescriptor)
6847        * 6.2.1. Lookup Credential Source by Credential ID algorithm
6848        * 6.2.2. The authenticatorMakeCredential operation
6849        * 6.2.3. The authenticatorGetAssertion operation
6850        * 6.3.1. Attested credential data
6851        * 7.1. Registering a new credential
6852        * 8.6. FIDO U2F Attestation Statement Format
6853        * 12.1. Registration
6854        * 12.3. Authentication (2) (3)
6855        * 13.3. credentialId Unsigned (2) (3)
6856
6857      #credential-public-keyReferenced in:
```

```
5976        * 4. Terminology (2) (3) (4) (5) (6) (7)
5977        * 5.2.1. Information about Public Key Credential (interface
5978          AuthenticatorAttestationResponse)
5979        * 6. WebAuthn Authenticator model
5980        * 6.3. Attestation (2) (3)
5981        * 6.3.1. Attested credential data (2)
5982        * 12.1. Registration (2)

5983
5984      #credential-key-pairReferenced in:
5985        * 4. Terminology (2) (3)
5986
5987      #credential-private-keyReferenced in:
5988        * 4. Terminology (2) (3) (4) (5) (6)
5989        * 5.1. PublicKeyCredential Interface
5990        * 5.2.2. Web Authentication Assertion (interface
5991          AuthenticatorAssertionResponse)
5992        * 6. WebAuthn Authenticator model
5993        * 6.2.2. The authenticatorGetAssertion operation
5994        * 6.3. Attestation (2)
5995        * 7.2. Verifying an authentication assertion
5996

5997      #public-key-credential-sourceReferenced in:
5998        * 4. Terminology (2) (3) (4) (5) (6) (7) (8)
5999        * 5.1.3. Create a new credential - PublicKeyCredential's
6000          [[Create]](origin, options, sameOriginWithAncestors) method

6001
6002      #public-key-credential-source-managing-authenticatorReferenced in:
6003        * 4. Terminology
6004
6005      #public-key-credentialReferenced in:
6006        * 1. Introduction (2) (3) (4) (5)
6007        * 4. Terminology (2) (3) (4) (5) (6) (7) (8)
6008        * 5. Web Authentication API (2) (3) (4)
6009        * 5.1. PublicKeyCredential Interface
6010        * 5.1.3. Create a new credential - PublicKeyCredential's
```

```
6858        * 4. Terminology (2) (3) (4) (5) (6) (7)
6859        * 5.2.1. Information about Public Key Credential (interface
6860          AuthenticatorAttestationResponse)
6861        * 6. WebAuthn Authenticator Model
6862        * 6.3. Attestation (2) (3)
6863        * 6.3.1. Attested credential data (2) (3)
6864        * 12.1. Registration (2)
6865        * 13.3. credentialId Unsigned
6866
6867      #credential-key-pairReferenced in:
6868        * 4. Terminology (2) (3)
6869
6870      #credential-private-keyReferenced in:
6871        * 4. Terminology (2) (3) (4) (5) (6)
6872        * 5.1. PublicKeyCredential Interface
6873        * 5.2.2. Web Authentication Assertion (interface
6874          AuthenticatorAssertionResponse)
6875        * 6. WebAuthn Authenticator Model

6876        * 6.3. Attestation (2)
6877        * 7.2. Verifying an authentication assertion
6878
6879      #human-palatabilityReferenced in:
6880        * 4. Terminology
6881        * 5.4.1. Public Key Entity Description (dictionary
6882          PublicKeyCredentialEntity) (2)
6883
6884      #public-key-credential-sourceReferenced in:
6885        * 4. Terminology (2) (3) (4) (5) (6) (7) (8) (9) (10) (11)
6886        * 5.1.3. Create a new credential - PublicKeyCredential's
6887          [[Create]](origin, options, sameOriginWithAncestors) method
6888        * 6. WebAuthn Authenticator Model
6889        * 6.2.1. Lookup Credential Source by Credential ID algorithm (2)
6890        * 6.2.2. The authenticatorMakeCredential operation
6891        * 6.2.3. The authenticatorGetAssertion operation (2)
6892
6893      #public-key-credential-source-typeReferenced in:
6894        * 6.2.2. The authenticatorMakeCredential operation (2)
6895
6896      #public-key-credential-source-idReferenced in:
6897        * 6.2.1. Lookup Credential Source by Credential ID algorithm (2)
6898        * 6.2.2. The authenticatorMakeCredential operation
6899        * 6.2.3. The authenticatorGetAssertion operation
6900
6901      #public-key-credential-source-privatekeyReferenced in:
6902        * 6.2.2. The authenticatorMakeCredential operation
6903        * 6.2.3. The authenticatorGetAssertion operation
6904
6905      #public-key-credential-source-rpidReferenced in:
6906        * 6. WebAuthn Authenticator Model
6907        * 6.2.2. The authenticatorMakeCredential operation
6908        * 6.2.3. The authenticatorGetAssertion operation
6909
6910      #public-key-credential-source-userhandleReferenced in:
6911        * 6. WebAuthn Authenticator Model
6912        * 6.2.2. The authenticatorMakeCredential operation
6913        * 6.2.3. The authenticatorGetAssertion operation (2)
6914
6915      #public-key-credential-source-otheruiReferenced in:
6916        * 6.2.2. The authenticatorMakeCredential operation
6917
6918      #public-key-credential-source-managing-authenticatorReferenced in:
6919        * 4. Terminology
6920
6921      #public-key-credentialReferenced in:
6922        * 1. Introduction (2) (3) (4) (5)
6923        * 4. Terminology (2) (3) (4) (5) (6) (7) (8)
6924        * 5. Web Authentication API (2) (3) (4)
6925        * 5.1. PublicKeyCredential Interface
6926        * 5.1.3. Create a new credential - PublicKeyCredential's
```

#relying-party-identifierReferenced in:

#relying-party-identifierReferenced in:

```
6125      * 5.5. Options for Assertion Generation (dictionary
6126        PublicKeyCredentialRequestOptions)
6127      * 6. WebAuthn Authenticator model
6128
6129      #rp-idReferenced in:
6130        * 4. Terminology (2) (3) (4) (5)
6131        * 5. Web Authentication API (2) (3) (4) (5)
6132        * 5.1.3. Create a new credential - PublicKeyCredential's
6133        [[Create]](origin, options, sameOriginWithAncestors) method (2)
6134        * 5.1.4.1. PublicKeyCredential's
6135        [[DiscoverFromExternalSource]](origin, options,
6136        sameOriginWithAncestors) method (2)
6137        * 5.4.2. RP Parameters for Credential Generation (dictionary
6138        PublicKeyCredentialRpEntity)
6139      * 6. WebAuthn Authenticator model
6140        * 6.1. Authenticator data (2) (3) (4) (5) (6)
6141        * 6.1.1. Signature Counter Considerations
6142      * 6.2.1. The authenticatorMakeCredential operation (2)
6143      * 6.2.2. The authenticatorGetAssertion operation (2)
6144      * 7.1. Registering a new credential (2)
6145      * 7.2. Verifying an authentication assertion
6146      * 8.4. Android Key Attestation Statement Format
6147      * 8.6. FIDO U2F Attestation Statement Format

6148
6149      #test-of-user-presenceReferenced in:
6150        * 4. Terminology (2) (3) (4) (5) (6)
6151      * 6.2.1. The authenticatorMakeCredential operation
6152      * 6.2.2. The authenticatorGetAssertion operation
6153      * 10.2. Simple Transaction Authorization Extension (txAuthSimple)
6154      * 10.3. Generic Transaction Authorization Extension (txAuthGeneric)
6155
6156      #user-consentReferenced in:
6157        * 1. Introduction (2)
6158        * 4. Terminology (2)
6159      * 5. Web Authentication API


6160      * 5.1.4. Use an existing credential to make an assertion -
6161        PublicKeyCredential's [[Get]](options) method




6162      * 5.2.2. Web Authentication Assertion (interface
6163        AuthenticatorAssertionResponse)
6164      * 5.4.6. Attestation Conveyance Preference enumeration (enum
6165        AttestationConveyancePreference)
6166      * 6. WebAuthn Authenticator model (2) (3)
6167      * 6.2.1. The authenticatorMakeCredential operation (2) (3) (4) (5)
6168        (6)
6169      * 6.2.2. The authenticatorGetAssertion operation (2) (3) (4) (5)
6170      * 11.2. WebAuthn Extension Identifier Registrations


6171
6172      #user-handleReferenced in:

6173      * 4. Terminology
6174      * 5.1.4.1. PublicKeyCredential's
6175        [[DiscoverFromExternalSource]](origin, options,
6176      sameOriginWithAncestors) method
6177      * 5.2.2. Web Authentication Assertion (interface
6178      AuthenticatorAssertionResponse)
6179      * 5.4. Options for Credential Creation (dictionary
6180      MakePublicKeyCredentialOptions)
6181      * 5.4.3. User Account Parameters for Credential Generation
6182        (dictionary PublicKeyCredentialUserEntity)
6183      * 6.2.1. The authenticatorMakeCredential operation
6184      * 6.2.2. The authenticatorGetAssertion operation
6185
```

```
7062      * 5.5. Options for Assertion Generation (dictionary
7063        PublicKeyCredentialRequestOptions)

7064
7065      #rp-idReferenced in:
7066        * 4. Terminology (2) (3) (4) (5)
7067        * 5. Web Authentication API (2) (3) (4) (5)
7068        * 5.1.3. Create a new credential - PublicKeyCredential's
7069        [[Create]](origin, options, sameOriginWithAncestors) method (2)
7070        * 5.1.4.1. PublicKeyCredential's
7071        [[DiscoverFromExternalSource]](origin, options,
7072        sameOriginWithAncestors) method (2)
7073        * 5.4.2. RP Parameters for Credential Generation (dictionary
7074        PublicKeyCredentialRpEntity)

7075      * 6.1. Authenticator data (2) (3) (4) (5) (6)
7076      * 6.1.1. Signature Counter Considerations
7077      * 6.2.2. The authenticatorMakeCredential operation (2) (3)
7078      * 6.2.3. The authenticatorGetAssertion operation (2)
7079      * 7.1. Registering a new credential (2)
7080      * 7.2. Verifying an authentication assertion
7081      * 8.4. Android Key Attestation Statement Format
7082      * 8.6. FIDO U2F Attestation Statement Format
7083      * 10.1. FIDO AppID Extension (appid)
7084
7085      #test-of-user-presenceReferenced in:
7086        * 4. Terminology (2) (3) (4) (5) (6)
7087      * 6.2.2. The authenticatorMakeCredential operation (2)
7088      * 6.2.3. The authenticatorGetAssertion operation
7089      * 10.2. Simple Transaction Authorization Extension (txAuthSimple)
7090      * 10.3. Generic Transaction Authorization Extension (txAuthGeneric)
7091
7092      #user-consentReferenced in:
7093        * 1. Introduction (2)
7094        * 4. Terminology (2)
7095        * 5. Web Authentication API
7096      * 5.1.3. Create a new credential - PublicKeyCredential's
7097      [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
7098      * 5.1.4. Use an existing credential to make an assertion -
7099        PublicKeyCredential's [[Get]](options) method
7100      * 5.1.4.1. PublicKeyCredential's
7101      [[DiscoverFromExternalSource]](origin, options,
7102      sameOriginWithAncestors) method
7103      * 5.2.2. Web Authentication Assertion (interface
7104      AuthenticatorAssertionResponse)
7105      * 5.4.6. Attestation Conveyance Preference enumeration (enum
7106      AttestationConveyancePreference)
7107      * 6. WebAuthn Authenticator Model (2) (3)
7108      * 6.2.2. The authenticatorMakeCredential operation (2) (3) (4) (5)
7109      (6) (7) (8)
7110      * 6.2.3. The authenticatorGetAssertion operation (2) (3) (4) (5)
7111      * 11.2. WebAuthn Extension Identifier Registrations
7112      * 14.2. Registration Ceremony Privacy (2)
7113      * 14.3. Authentication Ceremony Privacy (2) (3)
7114
7115      #user-handleReferenced in:
7116      * 2.2.1. Backwards Compatibility with FIDO U2F
7117      * 4. Terminology
7118      * 5.1.4.1. PublicKeyCredential's
7119      [[DiscoverFromExternalSource]](origin, options,
7120      sameOriginWithAncestors) method (2)
7121      * 5.2.2. Web Authentication Assertion (interface
7122      AuthenticatorAssertionResponse) (2)


7123      * 5.4.3. User Account Parameters for Credential Generation
7124        (dictionary PublicKeyCredentialUserEntity)
7125      * 6.2.2. The authenticatorMakeCredential operation

7126
```

```
#user-verificationReferenced in:
  * 1. Introduction
  * 4. Terminology (2) (3) (4) (5) (6) (7) (8) (9)
  * 5.1.3. Create a new credential - PublicKeyCredential's
    [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
  * 5.1.4.1. PublicKeyCredential's
    [[DiscoverFromExternalSource]](origin, options,
    sameOriginWithAncestors) method (2) (3)
  * 5.1.6. Availability of User-Verifying Platform Authenticator -
    PublicKeyCredential's
    isUserVerifyingPlatformAuthenticatorAvailable() method (2) (3) (4)
    (5)
  * 5.4.4. Authenticator Selection Criteria (dictionary
    AuthenticatorSelectionCriteria)
  * 5.5. Options for Assertion Generation (dictionary
    PublicKeyCredentialRequestOptions)
  * 5.8.6. User Verification Requirement enumeration (enum
    UserVerificationRequirement) (2) (3) (4)
  * 6.2.1. The authenticatorMakeCredential operation (2) (3)
  * 6.2.2. The authenticatorGetAssertion operation (2) (3)
  * 10.2. Simple Transaction Authorization Extension (txAuthSimple)
  * 10.3. Generic Transaction Authorization Extension (txAuthGeneric)
  * 12.2. Registration Specifically with User Verifying Platform
    Authenticator

#concept-user-presentReferenced in:
  * 4. Terminology
  * 6.1. Authenticator data (2) (3)


#upReferenced in:
  * 6.1. Authenticator data

#concept-user-verifiedReferenced in:
  * 4. Terminology
  * 6.1. Authenticator data (2) (3)


#uvReferenced in:
  * 5.8.6. User Verification Requirement enumeration (enum
    UserVerificationRequirement) (2)
  * 6.1. Authenticator data

#webauthn-clientReferenced in:
  * 4. Terminology (2)
  * 6.2.1. The authenticatorMakeCredential operation
  * 6.2.2. The authenticatorGetAssertion operation


#web-authentication-apiReferenced in:
  * 1. Introduction (2) (3)
  * 4. Terminology (2)

#publickeycredentialReferenced in:
  * 1. Introduction
  * 5.1. PublicKeyCredential Interface (2) (3) (4) (5) (6) (7) (8)
  * 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 -
```

```
#user-verificationReferenced in:
  * 1. Introduction
  * 4. Terminology (2) (3) (4) (5) (6) (7) (8) (9)
  * 5.1.3. Create a new credential - PublicKeyCredential's
    [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
  * 5.1.4.1. PublicKeyCredential's
    [[DiscoverFromExternalSource]](origin, options,
    sameOriginWithAncestors) method (2) (3)
  * 5.1.7. Availability of User-Verifying Platform Authenticator -
    PublicKeyCredential's
    isUserVerifyingPlatformAuthenticatorAvailable() method (2) (3) (4)
    (5)
  * 5.4.4. Authenticator Selection Criteria (dictionary
    AuthenticatorSelectionCriteria)
  * 5.5. Options for Assertion Generation (dictionary
    PublicKeyCredentialRequestOptions)
  * 5.10.6. User Verification Requirement enumeration (enum
    UserVerificationRequirement) (2) (3) (4)
  * 6.2.2. The authenticatorMakeCredential operation (2) (3)
  * 6.2.3. The authenticatorGetAssertion operation
  * 7.1. Registering a new credential (2)
  * 7.2. Verifying an authentication assertion (2)
  * 10.2. Simple Transaction Authorization Extension (txAuthSimple)
  * 10.3. Generic Transaction Authorization Extension (txAuthGeneric)
  * 12.2. Registration Specifically with User Verifying Platform
    Authenticator

#concept-user-presentReferenced in:
  * 4. Terminology
  * 6.1. Authenticator data (2) (3)
  * 7.1. Registering a new credential
  * 7.2. Verifying an authentication assertion

#upReferenced in:
  * 6.1. Authenticator data

#concept-user-verifiedReferenced in:
  * 4. Terminology
  * 6.1. Authenticator data (2) (3)
  * 7.1. Registering a new credential
  * 7.2. Verifying an authentication assertion

#uvReferenced in:
  * 5.10.6. User Verification Requirement enumeration (enum
    UserVerificationRequirement) (2)
  * 6.1. Authenticator data

#webauthn-clientReferenced in:
  * 4. Terminology (2) (3) (4)
  * 6.2. Authenticator operations
  * 6.2.2. The authenticatorMakeCredential operation
  * 6.2.3. The authenticatorGetAssertion operation
  * 13. Security Considerations

#web-authentication-apiReferenced in:
  * 1. Introduction (2) (3)
  * 4. Terminology (2) (3) (4)
  * 13. Security Considerations

#publickeycredentialReferenced in:
  * 1. Introduction
  * 5.1. PublicKeyCredential Interface (2) (3) (4) (5) (6) (7) (8)
  * 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
  * 5.1.5. Store an existing credential - PublicKeyCredential's
    [[Store]](credential, sameOriginWithAncestors) method (2)
  * 5.1.7. Availability of User-Verifying Platform Authenticator -
```

```
6247      PublicKeyCredential's
6248      isUserVerifyingPlatformAuthenticatorAvailable() method
6249    * 5.8.3. Credential Descriptor (dictionary
6250      PublicKeyCredentialDescriptor)
6251    * 7. Relying Party Operations
6252    * 7.2. Verifying an authentication assertion
6253
6254  #dom-publickeycredential-rawidReferenced in:
6255    * 5.1. PublicKeyCredential Interface
6256    * 7.2. Verifying an authentication assertion
6257
6258  #dom-publickeycredential-getclientextensionresultsReferenced in:
6259    * 5.1. PublicKeyCredential Interface
6260    * 9.4. Client extension processing
6261
6262  #dom-publickeycredential-responseReferenced in:
6263    * 5.1. PublicKeyCredential Interface
6264    * 5.1.3. Create a new credential - PublicKeyCredential's
6265      [[Create]](origin, options, sameOriginWithAncestors) method
6266    * 5.1.4.1. PublicKeyCredential's
6267      [[DiscoverFromExternalSource]](origin, options,
6268      sameOriginWithAncestors) method
6269    * 7.2. Verifying an authentication assertion
6270
6271  #dom-publickeycredential-identifier-slotReferenced in:
6272    * 5.1. PublicKeyCredential Interface (2)
6273    * 5.1.3. Create a new credential - PublicKeyCredential's
6274      [[Create]](origin, options, sameOriginWithAncestors) method
6275    * 5.1.4.1. PublicKeyCredential's
6276      [[DiscoverFromExternalSource]](origin, options,
6277      sameOriginWithAncestors) method
6278
6279  #dom-publickeycredential-clientextensionsresults-slotReferenced in:
6280    * 5.1. PublicKeyCredential Interface
6281    * 5.1.3. Create a new credential - PublicKeyCredential's
6282      [[Create]](origin, options, sameOriginWithAncestors) method
6283    * 5.1.4.1. PublicKeyCredential's
6284      [[DiscoverFromExternalSource]](origin, options,
6285      sameOriginWithAncestors) method
6286
6287  #dom-credentialcreationoptions-publickeyReferenced in:
6288    * 5.1.3. Create a new credential - PublicKeyCredential's
6289      [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
6290
6291  #dom-credentialrequestoptions-publickeyReferenced in:
6292    * 5.1.4.1. PublicKeyCredential's
6293      [[DiscoverFromExternalSource]](origin, options,
6294      sameOriginWithAncestors) method (2) (3)
6295
6296  #dom-publickeycredential-create-slotReferenced in:
6297    * 5.1. PublicKeyCredential Interface
6298    * 5.6. Abort operations with AbortSignal (2) (3) (4) (5)
6299    * 6.2.1. The authenticatorMakeCredential operation
6300
6301  #dom-publickeycredential-create-origin-options-sameoriginwithancestors-
6302  originReferenced in:
6303    * 5.1.3. Create a new credential - PublicKeyCredential's
6304      [[Create]](origin, options, sameOriginWithAncestors) method
6305
6306  #dom-publickeycredential-create-origin-options-sameoriginwithancestors-
6307  optionsReferenced in:
6308    * 7.1. Registering a new credential
6309
6310  #effective-user-verification-requirement-for-credential-creationReferen
6311  ced in:
6312    * 6.2.1. The authenticatorMakeCredential operation
```

```
7197      PublicKeyCredential's
7198      isUserVerifyingPlatformAuthenticatorAvailable() method
7199    * 5.10.3. Credential Descriptor (dictionary
7200      PublicKeyCredentialDescriptor)
7201    * 7. Relying Party Operations
7202    * 7.2. Verifying an authentication assertion
7203
7204  #dom-publickeycredential-rawidReferenced in:
7205    * 5.1. PublicKeyCredential Interface
7206    * 7.2. Verifying an authentication assertion
7207
7208  #dom-publickeycredential-getclientextensionresultsReferenced in:
7209    * 5.1. PublicKeyCredential Interface
7210    * 9.4. Client extension processing
7211
7212  #dom-publickeycredential-responseReferenced in:
7213    * 5.1. PublicKeyCredential Interface
7214    * 5.1.3. Create a new credential - PublicKeyCredential's
7215      [[Create]](origin, options, sameOriginWithAncestors) method
7216    * 5.1.4.1. PublicKeyCredential's
7217      [[DiscoverFromExternalSource]](origin, options,
7218      sameOriginWithAncestors) method
7219    * 7.2. Verifying an authentication assertion (2)
7220
7221  #dom-publickeycredential-identifier-slotReferenced in:
7222    * 5.1. PublicKeyCredential Interface (2)
7223    * 5.1.3. Create a new credential - PublicKeyCredential's
7224      [[Create]](origin, options, sameOriginWithAncestors) method
7225    * 5.1.4.1. PublicKeyCredential's
7226      [[DiscoverFromExternalSource]](origin, options,
7227      sameOriginWithAncestors) method
7228
7229  #dom-publickeycredential-clientextensionsresults-slotReferenced in:
7230    * 5.1. PublicKeyCredential Interface
7231    * 5.1.3. Create a new credential - PublicKeyCredential's
7232      [[Create]](origin, options, sameOriginWithAncestors) method
7233    * 5.1.4.1. PublicKeyCredential's
7234      [[DiscoverFromExternalSource]](origin, options,
7235      sameOriginWithAncestors) method
7236
7237  #dom-credentialcreationoptions-publickeyReferenced in:
7238    * 5.1.3. Create a new credential - PublicKeyCredential's
7239      [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
7240
7241  #dom-credentialrequestoptions-publickeyReferenced in:
7242    * 5.1.4.1. PublicKeyCredential's
7243      [[DiscoverFromExternalSource]](origin, options,
7244      sameOriginWithAncestors) method (2) (3)
7245
7246  #dom-publickeycredential-create-slotReferenced in:
7247    * 4. Terminology
7248    * 5.1. PublicKeyCredential Interface
7249    * 5.4.5. Authenticator Attachment enumeration (enum
7250      AuthenticatorAttachment)
7251    * 5.6. Abort operations with AbortSignal (2) (3) (4) (5)
7252    * 6.2.2. The authenticatorMakeCredential operation
7253    * 14.2. Registration Ceremony Privacy
7254
7255  #dom-publickeycredential-create-origin-options-sameoriginwithancestors-
7256  originReferenced in:
7257    * 5.1.3. Create a new credential - PublicKeyCredential's
7258      [[Create]](origin, options, sameOriginWithAncestors) method
7259
7260  #dom-publickeycredential-create-origin-options-sameoriginwithancestors-
7261  optionsReferenced in:
7262    * 7.1. Registering a new credential
7263
7264  #effective-user-verification-requirement-for-credential-creationReferen
7265  ced in:
7266    * 6.2.2. The authenticatorMakeCredential operation
```

#credentialcreationdata-attestationobjectresultReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's
[[Create]](origin, options, sameOriginWithAncestors) method

#credentialcreationdata-clientdatajsonresultReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's
[[Create]](origin, options, sameOriginWithAncestors) method

#credentialcreationdata-attestationconveyancepreferenceoptionReferenced
in:
* 5.1.3. Create a new credential - PublicKeyCredential's
[[Create]](origin, options, sameOriginWithAncestors) method

#credentialcreationdata-clientextensionresultsReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's
[[Create]](origin, options, sameOriginWithAncestors) method

#dom-publickeycredential-collectfromcredentialstore-slotReferenced in:
* 5.1.4. Use an existing credential to make an assertion -
PublicKeyCredential's [[Get]](options) method

#dom-publickeycredential-discoverfromexternalsource-slotReferenced in:

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

* 5.6. Abort operations with AbortSignal (2) (3) (4) (5)
* 6.2.2. The authenticatorGetAssertion operation

#dom-publickeycredential-discoverfromexternalsource-origin-options-same
originwithancestors-originReferenced in:
* 5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource]](origin, options,
sameOriginWithAncestors) method

#effective-user-verification-requirement-for-assertionReferenced in:
* 6.2.2. The authenticatorGetAssertion operation

#assertioncreationdata-credentialidresultReferenced in:
* 5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource]](origin, options,
sameOriginWithAncestors) method (2) (3)

#assertioncreationdata-clientdatajsonresultReferenced in:
* 5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource]](origin, options,
sameOriginWithAncestors) method

#assertioncreationdata-authenticatordataresultReferenced 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

#assertioncreationdata-clientextensionresultsReferenced in:

#credentialcreationdata-attestationobjectresultReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's
[[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
(4) (5)

#credentialcreationdata-clientdatajsonresultReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's
[[Create]](origin, options, sameOriginWithAncestors) method

#credentialcreationdata-attestationconveyancepreferenceoptionReferenced
in:
* 5.1.3. Create a new credential - PublicKeyCredential's
[[Create]](origin, options, sameOriginWithAncestors) method

#credentialcreationdata-clientextensionresultsReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's
[[Create]](origin, options, sameOriginWithAncestors) method

#dom-publickeycredential-collectfromcredentialstore-slotReferenced in:
* 5.1.4. Use an existing credential to make an assertion -
PublicKeyCredential's [[Get]](options) method

#dom-publickeycredential-discoverfromexternalsource-slotReferenced in:
* 4. Terminology
* 5.1. PublicKeyCredential Interface
* 5.1.4. Use an existing credential to make an assertion -
PublicKeyCredential's [[Get]](options) method
* 5.4.5. Authenticator Attachment enumeration (enum
AuthenticatorAttachment)
* 5.6. Abort operations with AbortSignal (2) (3) (4) (5)
* 6.2.3. The authenticatorGetAssertion operation
* 14.3. Authentication Ceremony Privacy

#dom-publickeycredential-discoverfromexternalsource-origin-options-same
originwithancestors-originReferenced in:
* 5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource]](origin, options,
sameOriginWithAncestors) method

#effective-user-verification-requirement-for-assertionReferenced in:
* 6.2.3. The authenticatorGetAssertion operation

#assertioncreationdata-credentialidresultReferenced in:
* 5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource]](origin, options,
sameOriginWithAncestors) method (2) (3)

#assertioncreationdata-clientdatajsonresultReferenced in:
* 5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource]](origin, options,
sameOriginWithAncestors) method

#assertioncreationdata-authenticatordataresultReferenced 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 (2) (3) (4)
* 6.2.3. The authenticatorGetAssertion operation

#assertioncreationdata-clientextensionresultsReferenced in:

```
6377      * 5.1.4.1. PublicKeyCredential's
6378        [[DiscoverFromExternalSource]](origin, options,
6379        sameOriginWithAncestors) method
6380
6381    #authenticatorresponseReferenced in:
6382      * 5.1. PublicKeyCredential Interface (2)
6383      * 5.2. Authenticator Responses (interface AuthenticatorResponse) (2)
6384      * 5.2.1. Information about Public Key Credential (interface
6385        AuthenticatorAttestationResponse) (2)
6386      * 5.2.2. Web Authentication Assertion (interface
6387        AuthenticatorAssertionResponse) (2)
6388
6389    #dom-authenticatorresponse-clientdatajsonReferenced in:
6390      * 5.1.3. Create a new credential - PublicKeyCredential's
6391        [[Create]](origin, options, sameOriginWithAncestors) method (2)
6392      * 5.1.4.1. PublicKeyCredential's
6393        [[DiscoverFromExternalSource]](origin, options,
6394        sameOriginWithAncestors) method (2)
6395      * 5.2. Authenticator Responses (interface AuthenticatorResponse)
6396      * 5.2.1. Information about Public Key Credential (interface
6397        AuthenticatorAttestationResponse)
6398      * 5.2.2. Web Authentication Assertion (interface
6399        AuthenticatorAssertionResponse)
6400      * 7.1. Registering a new credential (2)
6401      * 7.2. Verifying an authentication assertion
6402
6403    #authenticatorattestationresponseReferenced in:
6404      * 5.1. PublicKeyCredential Interface
6405      * 5.1.3. Create a new credential - PublicKeyCredential's
6406        [[Create]](origin, options, sameOriginWithAncestors) method
6407      * 5.2.1. Information about Public Key Credential (interface
6408        AuthenticatorAttestationResponse) (2)
6409      * 7. Relying Party Operations
6410      * 7.1. Registering a new credential (2) (3)
6411
6412    #dom-authenticatorattestationresponse-attestationobjectReferenced in:
6413      * 5.1.3. Create a new credential - PublicKeyCredential's
6414        [[Create]](origin, options, sameOriginWithAncestors) method
6415      * 5.2.1. Information about Public Key Credential (interface
6416        AuthenticatorAttestationResponse)
6417      * 7.1. Registering a new credential
6418
6419    #authenticatorassertionresponseReferenced in:
6420      * 4. Terminology
6421      * 5.1. PublicKeyCredential Interface
6422      * 5.1.4.1. PublicKeyCredential's
6423        [[DiscoverFromExternalSource]](origin, options,
6424        sameOriginWithAncestors) method
6425      * 5.2.2. Web Authentication Assertion (interface
6426        AuthenticatorAssertionResponse) (2)
6427      * 7. Relying Party Operations
6428
6429    #dom-authenticatorassertionresponse-authenticatordataReferenced in:
6430      * 5.1.4.1. PublicKeyCredential's
6431        [[DiscoverFromExternalSource]](origin, options,
6432        sameOriginWithAncestors) method
6433      * 5.2.2. Web Authentication Assertion (interface
6434        AuthenticatorAssertionResponse)
6435      * 7.2. Verifying an authentication assertion
6436
6437    #dom-authenticatorassertionresponse-signatureReferenced in:
6438      * 5.1.4.1. PublicKeyCredential's
6439        [[DiscoverFromExternalSource]](origin, options,
6440        sameOriginWithAncestors) method
6441      * 5.2.2. Web Authentication Assertion (interface
6442        AuthenticatorAssertionResponse)
6443      * 7.2. Verifying an authentication assertion
6444
6445    #dom-authenticatorassertionresponse-userhandleReferenced in:
```

```
7337      * 5.1.4.1. PublicKeyCredential's
7338        [[DiscoverFromExternalSource]](origin, options,
7339        sameOriginWithAncestors) method
7340
7341    #authenticatorresponseReferenced in:
7342      * 5.1. PublicKeyCredential Interface (2)
7343      * 5.2. Authenticator Responses (interface AuthenticatorResponse) (2)
7344      * 5.2.1. Information about Public Key Credential (interface
7345        AuthenticatorAttestationResponse) (2)
7346      * 5.2.2. Web Authentication Assertion (interface
7347        AuthenticatorAssertionResponse) (2)
7348
7349    #dom-authenticatorresponse-clientdatajsonReferenced in:
7350      * 5.1.3. Create a new credential - PublicKeyCredential's
7351        [[Create]](origin, options, sameOriginWithAncestors) method (2)
7352      * 5.1.4.1. PublicKeyCredential's
7353        [[DiscoverFromExternalSource]](origin, options,
7354        sameOriginWithAncestors) method (2)
7355      * 5.2. Authenticator Responses (interface AuthenticatorResponse)
7356      * 5.2.1. Information about Public Key Credential (interface
7357        AuthenticatorAttestationResponse)
7358      * 5.2.2. Web Authentication Assertion (interface
7359        AuthenticatorAssertionResponse)
7360      * 7.1. Registering a new credential (2)
7361      * 7.2. Verifying an authentication assertion
7362
7363    #authenticatorattestationresponseReferenced in:
7364      * 5.1. PublicKeyCredential Interface
7365      * 5.1.3. Create a new credential - PublicKeyCredential's
7366        [[Create]](origin, options, sameOriginWithAncestors) method
7367      * 5.2.1. Information about Public Key Credential (interface
7368        AuthenticatorAttestationResponse) (2)
7369      * 7. Relying Party Operations
7370      * 7.1. Registering a new credential (2)
7371
7372    #dom-authenticatorattestationresponse-attestationobjectReferenced in:
7373      * 5.1.3. Create a new credential - PublicKeyCredential's
7374        [[Create]](origin, options, sameOriginWithAncestors) method
7375      * 5.2.1. Information about Public Key Credential (interface
7376        AuthenticatorAttestationResponse)
7377      * 7.1. Registering a new credential
7378
7379    #authenticatorassertionresponseReferenced in:
7380      * 4. Terminology
7381      * 5.1. PublicKeyCredential Interface
7382      * 5.1.4.1. PublicKeyCredential's
7383        [[DiscoverFromExternalSource]](origin, options,
7384        sameOriginWithAncestors) method
7385      * 5.2.2. Web Authentication Assertion (interface
7386        AuthenticatorAssertionResponse) (2)
7387      * 7. Relying Party Operations
7388
7389    #dom-authenticatorassertionresponse-authenticatordataReferenced in:
7390      * 5.1.4.1. PublicKeyCredential's
7391        [[DiscoverFromExternalSource]](origin, options,
7392        sameOriginWithAncestors) method
7393      * 5.2.2. Web Authentication Assertion (interface
7394        AuthenticatorAssertionResponse)
7395      * 7.2. Verifying an authentication assertion
7396
7397    #dom-authenticatorassertionresponse-signatureReferenced in:
7398      * 5.1.4.1. PublicKeyCredential's
7399        [[DiscoverFromExternalSource]](origin, options,
7400        sameOriginWithAncestors) method
7401      * 5.2.2. Web Authentication Assertion (interface
7402        AuthenticatorAssertionResponse)
7403      * 7.2. Verifying an authentication assertion
7404
7405    #dom-authenticatorassertionresponse-userhandleReferenced in:
7406      * 2.2.1. Backwards Compatibility with FIDO U2F
```

```
      * 5.1.4.1. PublicKeyCredential's
        [[DiscoverFromExternalSource]](origin, options,
        sameOriginWithAncestors) method
      * 5.2.2. Web Authentication Assertion (interface
        AuthenticatorAssertionResponse)


    #dictdef-publickeycredentialparametersReferenced in:
      * 5.3. Parameters for Credential Generation (dictionary
        PublicKeyCredentialParameters)
      * 5.4. Options for Credential Creation (dictionary
        MakePublicKeyCredentialOptions) (2)

    #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
        MakePublicKeyCredentialOptions)

    #dom-makepublickeycredentialoptions-rpReferenced in:
      * 5.1.3. Create a new credential - PublicKeyCredential's
        [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
        (4) (5) (6)
      * 5.4. Options for Credential Creation (dictionary
        MakePublicKeyCredentialOptions)

    #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
        MakePublicKeyCredentialOptions)
      * 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
        MakePublicKeyCredentialOptions)


    #dom-makepublickeycredentialoptions-pubkeycredparamsReferenced in:
      * 5.1.3. Create a new credential - PublicKeyCredential's
        [[Create]](origin, options, sameOriginWithAncestors) method (2)
      * 5.4. Options for Credential Creation (dictionary
        MakePublicKeyCredentialOptions)

    #dom-makepublickeycredentialoptions-timeoutReferenced in:
      * 5.1.3. Create a new credential - PublicKeyCredential's
        [[Create]](origin, options, sameOriginWithAncestors) method (2)
      * 5.4. Options for Credential Creation (dictionary
        MakePublicKeyCredentialOptions)

    #dom-makepublickeycredentialoptions-excludecredentialsReferenced in:

      * 5.1.3. Create a new credential - PublicKeyCredential's
        [[Create]](origin, options, sameOriginWithAncestors) method
      * 5.4. Options for Credential Creation (dictionary
```



```
      * 5.1.4.1. PublicKeyCredential's
        [[DiscoverFromExternalSource]](origin, options,
        sameOriginWithAncestors) method
      * 5.2.2. Web Authentication Assertion (interface
        AuthenticatorAssertionResponse)
    * 7.2. Verifying an authentication assertion

    #dictdef-publickeycredentialparametersReferenced in:
      * 5.3. Parameters for Credential Generation (dictionary
        PublicKeyCredentialParameters)
      * 5.4. Options for Credential Creation (dictionary
        PublicKeyCredentialCreationOptions) (2)

    #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-publickeycredentialcreationoptionsReferenced in:
      * 5.1.1. CredentialCreationOptions Dictionary Extension
      * 5.1.3. Create a new credential - PublicKeyCredential's
        [[Create]](origin, options, sameOriginWithAncestors) method
      * 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)
        (4) (5) (6)
      * 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-publickeycredentialcreationoptions-pubkeycredparamsReferenced in:
      * 5.1.3. Create a new credential - PublicKeyCredential's
        [[Create]](origin, options, sameOriginWithAncestors) method (2)
      * 5.4. Options for Credential Creation (dictionary
        PublicKeyCredentialCreationOptions)

    #dom-publickeycredentialcreationoptions-timeoutReferenced in:
      * 5.1.3. Create a new credential - PublicKeyCredential's
        [[Create]](origin, options, sameOriginWithAncestors) method (2)
      * 5.4. Options for Credential Creation (dictionary
        PublicKeyCredentialCreationOptions)

    #dom-publickeycredentialcreationoptions-excludecredentialsReferenced
    in:
      * 5.1.3. Create a new credential - PublicKeyCredential's
        [[Create]](origin, options, sameOriginWithAncestors) method
      * 5.4. Options for Credential Creation (dictionary
```

```
6513      MakePublicKeyCredentialOptions)
6514
6515    #dom-makepublickeycredentialoptions-authenticatorselectionReferenced
6516    in:
6517      * 5.1.3. Create a new credential - PublicKeyCredential's
6518        [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
6519        (4) (5) (6)
6520      * 5.4. Options for Credential Creation (dictionary
6521        MakePublicKeyCredentialOptions)
6522      * 6.2.1. The authenticatorMakeCredential operation
6523
6524    #dom-makepublickeycredentialoptions-attestationReferenced in:
6525      * 5.1.3. Create a new credential - PublicKeyCredential's
6526        [[Create]](origin, options, sameOriginWithAncestors) method
6527      * 5.4. Options for Credential Creation (dictionary
6528        MakePublicKeyCredentialOptions)
6529
6530    #dom-makepublickeycredentialoptions-extensionsReferenced in:
6531      * 5.1.3. Create a new credential - PublicKeyCredential's
6532        [[Create]](origin, options, sameOriginWithAncestors) method (2)
6533      * 5.4. Options for Credential Creation (dictionary
6534        MakePublicKeyCredentialOptions)


6535      * 9.3. Extending request parameters
6536
6537    #dictdef-publickeycredentialentityReferenced in:
6538      * 5.4.1. Public Key Entity Description (dictionary
6539        PublicKeyCredentialEntity) (2)
6540      * 5.4.2. RP Parameters for Credential Generation (dictionary
6541        PublicKeyCredentialRpEntity)
6542      * 5.4.3. User Account Parameters for Credential Generation
6543        (dictionary PublicKeyCredentialUserEntity)
6544
6545    #dom-publickeycredentialentity-nameReferenced in:
6546      * 5.4. Options for Credential Creation (dictionary
6547        MakePublicKeyCredentialOptions) (2)
6548      * 5.4.1. Public Key Entity Description (dictionary
6549        PublicKeyCredentialEntity)
6550      * 6.2.1. The authenticatorMakeCredential operation (2)
6551
6552    #dom-publickeycredentialentity-iconReferenced in:
6553      * 5.4.1. Public Key Entity Description (dictionary
6554        PublicKeyCredentialEntity)
6555
6556    #dictdef-publickeycredentialrpentityReferenced in:
6557      * 5.4. Options for Credential Creation (dictionary
6558        MakePublicKeyCredentialOptions) (2)


6559      * 5.4.2. RP Parameters for Credential Generation (dictionary
6560        PublicKeyCredentialRpEntity) (2)
6561      * 6.2.1. The authenticatorMakeCredential operation
6562
6563    #dom-publickeycredentialrpentity-idReferenced in:




6564      * 5.4. Options for Credential Creation (dictionary
6565        MakePublicKeyCredentialOptions)
6566      * 5.4.2. RP Parameters for Credential Generation (dictionary
6567        PublicKeyCredentialRpEntity)
6568      * 6.2.1. The authenticatorMakeCredential operation (2) (3) (4)
6569
6570    #dictdef-publickeycredentialuserentityReferenced in:
6571      * 5.4. Options for Credential Creation (dictionary
6572        MakePublicKeyCredentialOptions) (2)


6573      * 5.4.3. User Account Parameters for Credential Generation
```

```
7477      PublicKeyCredentialCreationOptions)
7478
7479    #dom-publickeycredentialcreationoptions-authenticatorselectionReference
7480    d in:
7481      * 5.1.3. Create a new credential - PublicKeyCredential's
7482        [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
7483        (4) (5) (6)
7484      * 5.4. Options for Credential Creation (dictionary
7485        PublicKeyCredentialCreationOptions)
7486      * 6.2.2. The authenticatorMakeCredential operation
7487
7488    #dom-publickeycredentialcreationoptions-attestationReferenced in:
7489      * 5.1.3. Create a new credential - PublicKeyCredential's
7490        [[Create]](origin, options, sameOriginWithAncestors) method
7491      * 5.4. Options for Credential Creation (dictionary
7492        PublicKeyCredentialCreationOptions)
7493
7494    #dom-publickeycredentialcreationoptions-extensionsReferenced in:
7495      * 5.1.3. Create a new credential - PublicKeyCredential's
7496        [[Create]](origin, options, sameOriginWithAncestors) method (2)
7497      * 5.4. Options for Credential Creation (dictionary
7498        PublicKeyCredentialCreationOptions)
7499      * 7.1. Registering a new credential (2)
7500      * 7.2. Verifying an authentication assertion
7501      * 9.3. Extending request parameters
7502
7503    #dictdef-publickeycredentialentityReferenced in:
7504      * 5.4.1. Public Key Entity Description (dictionary
7505        PublicKeyCredentialEntity) (2) (3)
7506      * 5.4.2. RP Parameters for Credential Generation (dictionary
7507        PublicKeyCredentialRpEntity)
7508      * 5.4.3. User Account Parameters for Credential Generation
7509        (dictionary PublicKeyCredentialUserEntity)
7510
7511    #dom-publickeycredentialentity-nameReferenced in:
7512      * 5.4. Options for Credential Creation (dictionary
7513        PublicKeyCredentialCreationOptions) (2)
7514      * 5.4.1. Public Key Entity Description (dictionary
7515        PublicKeyCredentialEntity) (2) (3) (4)
7516      * 6.2.2. The authenticatorMakeCredential operation (2)
7517
7518    #dom-publickeycredentialentity-iconReferenced in:
7519      * 5.4.1. Public Key Entity Description (dictionary
7520        PublicKeyCredentialEntity)
7521
7522    #dictdef-publickeycredentialrpentityReferenced in:
7523      * 5.4. Options for Credential Creation (dictionary
7524        PublicKeyCredentialCreationOptions) (2)
7525      * 5.4.1. Public Key Entity Description (dictionary
7526        PublicKeyCredentialEntity)
7527      * 5.4.2. RP Parameters for Credential Generation (dictionary
7528        PublicKeyCredentialRpEntity) (2)
7529      * 6.2.2. The authenticatorMakeCredential operation
7530
7531    #dom-publickeycredentialrpentity-idReferenced in:
7532      * 5.1.3. Create a new credential - PublicKeyCredential's
7533        [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
7534        (4) (5)
7535      * 5.4. Options for Credential Creation (dictionary
7536        PublicKeyCredentialCreationOptions)
7537      * 5.4.2. RP Parameters for Credential Generation (dictionary
7538        PublicKeyCredentialRpEntity)
7539      * 6.2.2. The authenticatorMakeCredential operation (2) (3) (4)
7540
7541    #dictdef-publickeycredentialuserentityReferenced in:
7542      * 5.4. Options for Credential Creation (dictionary
7543        PublicKeyCredentialCreationOptions) (2)
7544      * 5.4.1. Public Key Entity Description (dictionary
7545        PublicKeyCredentialEntity) (2)
7546      * 5.4.3. User Account Parameters for Credential Generation
```

(dictionary PublicKeyCredentialUserEntity) (2)
* 6.2.1. The authenticatorMakeCredential operation

#dom-publickeycredentialuserentity-idReferenced in:
* 5.4. Options for Credential Creation (dictionary MakePublicKeyCredentialOptions)
* 5.4.3. User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity)
* 6.2.1. The authenticatorMakeCredential operation

#dom-publickeycredentialuserentity-displaynameReferenced in:
* 4. Terminology
* 5.4. Options for Credential Creation (dictionary MakePublicKeyCredentialOptions)

* 5.4.3. User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity)
* 6.2.1. The authenticatorMakeCredential operation

#dictdef-authenticatorselectioncriteriaReferenced in:
* 5.4. Options for Credential Creation (dictionary MakePublicKeyCredentialOptions) (2)
* 5.4.4. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria) (2)

#dom-authenticatorselectioncriteria-authenticatorattachmentReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
* 5.4.4. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria)

#dom-authenticatorselectioncriteria-requireresidentkeyReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2)
* 5.4.4. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria)
* 6.2.1. The authenticatorMakeCredential operation

#dom-authenticatorselectioncriteria-userverificationReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2)
* 5.4.4. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria)

#enumdef-authenticatorattachmentReferenced in:
* 5.4.4. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria) (2)
* 5.4.5. Authenticator Attachment enumeration (enum AuthenticatorAttachment) (2)

#platform-authenticatorsReferenced in:
* 5.1.6. Availability of User-Verifying Platform Authenticator - PublicKeyCredential's isUserVerifyingPlatformAuthenticatorAvailable() method (2) (3) (4) (5)
* 5.4.5. Authenticator Attachment enumeration (enum AuthenticatorAttachment) (2)
* 12.1. Registration
* 12.2. Registration Specifically with User Verifying Platform Authenticator (2)

#roaming-authenticatorsReferenced in:
* 1.1.3. Other use cases and configurations

(dictionary PublicKeyCredentialUserEntity) (2)
* 6.2.2. The authenticatorMakeCredential operation

#dom-publickeycredentialuserentity-idReferenced in:
* 5.4. Options for Credential Creation (dictionary PublicKeyCredentialCreationOptions)
* 5.4.3. User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity)
* 6.2.2. The authenticatorMakeCredential operation

#dom-publickeycredentialuserentity-displaynameReferenced in:
* 4. Terminology
* 5.4. Options for Credential Creation (dictionary PublicKeyCredentialCreationOptions)
* 5.4.1. Public Key Entity Description (dictionary PublicKeyCredentialEntity)
* 5.4.3. User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity) (2) (3)
* 6.2.2. The authenticatorMakeCredential operation

#dictdef-authenticatorselectioncriteriaReferenced in:
* 5.4. Options for Credential Creation (dictionary PublicKeyCredentialCreationOptions) (2)
* 5.4.4. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria) (2)

#dom-authenticatorselectioncriteria-authenticatorattachmentReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
* 5.4.4. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria)

#dom-authenticatorselectioncriteria-requireresidentkeyReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2)
* 5.4.4. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria)
* 6.2.2. The authenticatorMakeCredential operation

#dom-authenticatorselectioncriteria-userverificationReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2)
* 5.4.4. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria)

#enumdef-authenticatorattachmentReferenced in:
* 5.4.4. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria) (2)
* 5.4.5. Authenticator Attachment enumeration (enum AuthenticatorAttachment) (2)

#attachment-modalityReferenced in:
* 5.4.5. Authenticator Attachment enumeration (enum AuthenticatorAttachment) (2)

#platform-authenticatorsReferenced in:
* 5.1.7. Availability of User-Verifying Platform Authenticator - PublicKeyCredential's isUserVerifyingPlatformAuthenticatorAvailable() method (2) (3) (4) (5)
* 5.4.5. Authenticator Attachment enumeration (enum AuthenticatorAttachment) (2)
* 12.1. Registration
* 12.2. Registration Specifically with User Verifying Platform Authenticator (2)
* 14.2. Registration Ceremony Privacy

#roaming-authenticatorsReferenced in:
* 1.1.3. Other use cases and configurations

```
6637      * 5.4.5. Authenticator Attachment enumeration (enum
6638        AuthenticatorAttachment) (2)
6639      * 12.1. Registration
6640
6641    #platform-attachmentReferenced in:
6642      * 5.4.5. Authenticator Attachment enumeration (enum
6643        AuthenticatorAttachment)
6644
6645    #cross-platform-attachedReferenced in:
6646      * 5.4.5. Authenticator Attachment enumeration (enum
6647        AuthenticatorAttachment) (2)
6648
6649    #attestation-conveyanceReferenced in:
6650      * 4. Terminology
6651      * 5.4. Options for Credential Creation (dictionary
6652        MakePublicKeyCredentialOptions)
6653      * 5.4.6. Attestation Conveyance Preference enumeration (enum
6654        AttestationConveyancePreference)
6655
6656    #enumdef-attestationconveyancepreferenceReferenced in:
6657      * 5.4. Options for Credential Creation (dictionary
6658        MakePublicKeyCredentialOptions) (2)
6659      * 5.4.6. Attestation Conveyance Preference enumeration (enum
6660        AttestationConveyancePreference) (2)
6661
6662    #dom-attestationconveyancepreference-noneReferenced in:
6663      * 5.4. Options for Credential Creation (dictionary
6664        MakePublicKeyCredentialOptions)
6665      * 5.4.6. Attestation Conveyance Preference enumeration (enum
6666        AttestationConveyancePreference)
6667
6668    #dom-attestationconveyancepreference-indirectReferenced in:
6669      * 5.4.6. Attestation Conveyance Preference enumeration (enum
6670        AttestationConveyancePreference)
6671
6672    #dom-attestationconveyancepreference-directReferenced in:
6673      * 5.4.6. Attestation Conveyance Preference enumeration (enum
6674        AttestationConveyancePreference)
6675
6676    #dictdef-publickeycredentialrequestoptionsReferenced in:
6677      * 5.1.2. CredentialRequestOptions Extension
6678      * 5.1.4.1. PublicKeyCredential's
6679        [[DiscoverFromExternalSource]](origin, options,
6680        sameOriginWithAncestors) method
6681      * 5.5. Options for Assertion Generation (dictionary
6682        PublicKeyCredentialRequestOptions) (2)
6683      * 7.2. Verifying an authentication assertion
6684
6685    #dom-publickeycredentialrequestoptions-challengeReferenced in:
6686      * 5.1.4.1. PublicKeyCredential's
6687        [[DiscoverFromExternalSource]](origin, options,
6688        sameOriginWithAncestors) method
6689      * 5.5. Options for Assertion Generation (dictionary
6690        PublicKeyCredentialRequestOptions) (2)
6691      * 13.1. Cryptographic Challenges
6692
6693    #dom-publickeycredentialrequestoptions-timeoutReferenced in:
6694      * 5.1.4.1. PublicKeyCredential's
6695        [[DiscoverFromExternalSource]](origin, options,
6696        sameOriginWithAncestors) method (2)
6697      * 5.5. Options for Assertion Generation (dictionary
6698        PublicKeyCredentialRequestOptions)
```

```
7617      * 5.4.5. Authenticator Attachment enumeration (enum
7618        AuthenticatorAttachment) (2)
7619      * 12.1. Registration
7620
7621    #platform-attachmentReferenced in:
7622      * 5.4.5. Authenticator Attachment enumeration (enum
7623        AuthenticatorAttachment)
7624
7625    #platform-credentialReferenced in:
7626      * 5.4.5. Authenticator Attachment enumeration (enum
7627        AuthenticatorAttachment) (2)
7628
7629    #cross-platform-attachedReferenced in:
7630      * 5.4.5. Authenticator Attachment enumeration (enum
7631        AuthenticatorAttachment) (2)
7632
7633    #roaming-credentialReferenced in:
7634      * 5.4.5. Authenticator Attachment enumeration (enum
7635        AuthenticatorAttachment) (2)
7636
7637    #attestation-conveyanceReferenced in:
7638      * 4. Terminology
7639      * 5.4. Options for Credential Creation (dictionary
7640        PublicKeyCredentialCreationOptions)
7641      * 5.4.6. Attestation Conveyance Preference enumeration (enum
7642        AttestationConveyancePreference)
7643
7644    #enumdef-attestationconveyancepreferenceReferenced in:
7645      * 5.4. Options for Credential Creation (dictionary
7646        PublicKeyCredentialCreationOptions) (2)
7647      * 5.4.6. Attestation Conveyance Preference enumeration (enum
7648        AttestationConveyancePreference) (2)
7649
7650    #dom-attestationconveyancepreference-noneReferenced in:
7651      * 5.4. Options for Credential Creation (dictionary
7652        PublicKeyCredentialCreationOptions)
7653      * 5.4.6. Attestation Conveyance Preference enumeration (enum
7654        AttestationConveyancePreference)
7655
7656    #dom-attestationconveyancepreference-indirectReferenced in:
7657      * 5.4.6. Attestation Conveyance Preference enumeration (enum
7658        AttestationConveyancePreference)
7659
7660    #dom-attestationconveyancepreference-directReferenced in:
7661      * 5.4.6. Attestation Conveyance Preference enumeration (enum
7662        AttestationConveyancePreference)
7663
7664    #dictdef-publickeycredentialrequestoptionsReferenced in:
7665      * 5.1.2. CredentialRequestOptions Dictionary Extension
7666      * 5.1.4.1. PublicKeyCredential's
7667        [[DiscoverFromExternalSource]](origin, options,
7668        sameOriginWithAncestors) method
7669      * 5.5. Options for Assertion Generation (dictionary
7670        PublicKeyCredentialRequestOptions) (2)
7671      * 7.2. Verifying an authentication assertion
7672
7673    #dom-publickeycredentialrequestoptions-challengeReferenced in:
7674      * 5.1.4.1. PublicKeyCredential's
7675        [[DiscoverFromExternalSource]](origin, options,
7676        sameOriginWithAncestors) method
7677      * 5.5. Options for Assertion Generation (dictionary
7678        PublicKeyCredentialRequestOptions) (2)
7679      * 13.1. Cryptographic Challenges
7680
7681    #dom-publickeycredentialrequestoptions-timeoutReferenced in:
7682      * 5.1.4.1. PublicKeyCredential's
7683        [[DiscoverFromExternalSource]](origin, options,
7684        sameOriginWithAncestors) method (2)
7685      * 5.5. Options for Assertion Generation (dictionary
7686        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)
* 10.1. FIDO AppId Extension (appid)

#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)

#dom-publickeycredentialrequestoptions-userverificationReferenced in:
* 5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method (2)
* 5.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)

#dom-publickeycredentialrequestoptions-extensionsReferenced in:
* 5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method (2)
* 5.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)

#typedefdef-authenticationextensionsReferenced in:
* 5.1. PublicKeyCredential Interface
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
* 5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method
* 5.4. Options for Credential Creation (dictionary MakePublicKeyCredentialOptions) (2)
* 5.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions) (2)
* 5.8.1. Client data used in WebAuthn signatures (dictionary CollectedClientData) (2)

#dictdef-collectedclientdataReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
* 5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options,

#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)
* 7.2. Verifying an authentication assertion (2)
* 14.3. Authentication Ceremony Privacy (2)

#dom-publickeycredentialrequestoptions-userverificationReferenced in:
* 5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method (2)
* 5.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)

#dom-publickeycredentialrequestoptions-extensionsReferenced in:
* 5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method (2)
* 5.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)
* 7.2. Verifying an authentication assertion

#dictdef-authenticationextensionsclientinputsReferenced in:
* 5.4. Options for Credential Creation (dictionary PublicKeyCredentialCreationOptions) (2)
* 5.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions) (2)
* 10.1. FIDO AppID Extension (appid)
* 10.2. Simple Transaction Authorization Extension (txAuthSimple)
* 10.3. Generic Transaction Authorization Extension (txAuthGeneric)
* 10.4. Authenticator Selection Extension (authnSel)
* 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)

#dictdef-authenticationextensionsclientoutputsReferenced in:
* 5.1. PublicKeyCredential Interface
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
* 5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method
* 7.1. Registering a new credential
* 7.2. Verifying an authentication assertion
* 10.1. FIDO AppID Extension (appid)
* 10.2. Simple Transaction Authorization Extension (txAuthSimple)
* 10.3. Generic Transaction Authorization Extension (txAuthGeneric)
* 10.4. Authenticator Selection Extension (authnSel)
* 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)

#dictdef-collectedclientdataReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
* 5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options,

sameOriginWithAncestors) method
* 5.8.1. Client data used in WebAuthn signatures (dictionary
CollectedClientData) (2)

#client-dataReferenced in:
* 5.2. Authenticator Responses (interface AuthenticatorResponse)
* 6. WebAuthn Authenticator model (2) (3) (4)
* 6.1. Authenticator data (2)
* 7.1. Registering a new credential
* 7.2. Verifying an authentication assertion
* 9. WebAuthn Extensions
* 9.4. Client extension processing
* 9.6. Example Extension

#dom-collectedclientdata-typeReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's
[[Create]](origin, options, sameOriginWithAncestors) method
* 5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource]](origin, options,
sameOriginWithAncestors) method
* 5.8.1. Client data used in WebAuthn signatures (dictionary
CollectedClientData)

* 7.1. Registering a new credential
* 7.2. Verifying an authentication assertion

#dom-collectedclientdata-challengeReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's
[[Create]](origin, options, sameOriginWithAncestors) method
* 5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource]](origin, options,
sameOriginWithAncestors) method
* 5.8.1. Client data used in WebAuthn signatures (dictionary
CollectedClientData)
* 7.1. Registering a new credential
* 7.2. Verifying an authentication assertion

#dom-collectedclientdata-originReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's
[[Create]](origin, options, sameOriginWithAncestors) method
* 5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource]](origin, options,
sameOriginWithAncestors) method
* 5.8.1. Client data used in WebAuthn signatures (dictionary
CollectedClientData)
* 7.1. Registering a new credential
* 7.2. Verifying an authentication assertion

#dom-collectedclientdata-hashalgorithmReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's
[[Create]](origin, options, sameOriginWithAncestors) method
* 5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource]](origin, options,
sameOriginWithAncestors) method
* 5.8.1. Client data used in WebAuthn signatures (dictionary
CollectedClientData) (2)
* 7.1. Registering a new credential
* 7.2. Verifying an authentication assertion

#dom-collectedclientdata-tokenbindingidReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's
[[Create]](origin, options, sameOriginWithAncestors) method
* 5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource]](origin, options,
sameOriginWithAncestors) method
* 5.8.1. Client data used in WebAuthn signatures (dictionary
CollectedClientData)
* 7.1. Registering a new credential
* 7.2. Verifying an authentication assertion

sameOriginWithAncestors) method
* 5.10.1. Client data used in WebAuthn signatures (dictionary
CollectedClientData) (2)

#client-dataReferenced in:
* 5.2. Authenticator Responses (interface AuthenticatorResponse)
* 6. WebAuthn Authenticator Model (2) (3) (4)
* 6.1. Authenticator data (2)
* 7.1. Registering a new credential
* 7.2. Verifying an authentication assertion
* 9. WebAuthn Extensions
* 9.4. Client extension processing

#dictdef-tokenbindingReferenced in:
* 5.10.1. Client data used in WebAuthn signatures (dictionary
CollectedClientData)

#dom-tokenbinding-statusReferenced in:
* 7.1. Registering a new credential
* 7.2. Verifying an authentication assertion

#dom-tokenbinding-idReferenced in:
* 7.1. Registering a new credential
* 7.2. Verifying an authentication assertion

#enumdef-tokenbindingstatusReferenced in:
* 5.10.1. Client data used in WebAuthn signatures (dictionary
CollectedClientData)

#dom-collectedclientdata-typeReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's
[[Create]](origin, options, sameOriginWithAncestors) method
* 5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource]](origin, options,
sameOriginWithAncestors) method
* 5.10.1. Client data used in WebAuthn signatures (dictionary
CollectedClientData)
* 7.1. Registering a new credential
* 7.2. Verifying an authentication assertion

#dom-collectedclientdata-challengeReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's
[[Create]](origin, options, sameOriginWithAncestors) method
* 5.1.4.1. PublicKeyCredential's
[[DiscoverFromExternalSource]](origin, options,
sameOriginWithAncestors) method
* 5.10.1. Client data used in WebAuthn signatures (dictionary
CollectedClientData)
* 7.1. Registering a new credential
* 7.2. Verifying an authentication assertion

/Users/jehodges/Documents/work/standards/W3C/webauthn/index-master-tr-5e63e57-WD-07.txt, Top line: 6816

#dom-collectedclientdata-clientextensionsReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
* 5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method
* 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

#dom-collectedclientdata-authenticatorextensionsReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
* 5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method
* 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 [[Create]](origin, options, sameOriginWithAncestors) method
* 5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method
* 5.2. Authenticator Responses (interface AuthenticatorResponse)
* 5.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse) (2)
* 5.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
* 5.8.1. Client data used in WebAuthn signatures (dictionary CollectedClientData)

#collectedclientdata-hash-of-the-serialized-client-dataReferenced 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)
* 5.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)
* 5.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
* 5.8.1. Client data used in WebAuthn signatures (dictionary CollectedClientData)
* 6. WebAuthn Authenticator model
* 6.2.1. The authenticatorMakeCredential operation
* 6.2.2. The authenticatorGetAssertion operation (2)
* 6.3.2. Attestation Statement Formats (2)
* 6.3.4. Generating an Attestation Object
* 7.1. Registering a new credential
* 8.2. Packed Attestation Statement Format
* 8.3. TPM Attestation Statement Format
* 8.4. Android Key Attestation Statement Format
* 8.5. Android SafetyNet Attestation Statement Format
* 8.6. FIDO U2F Attestation Statement Format

#enumdef-publickeycredentialtypeReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2)
* 5.3. Parameters for Credential Generation (dictionary PublicKeyCredentialParameters)
* 5.8.2. Credential Type enumeration (enum PublicKeyCredentialType)
* 5.8.3. Credential Descriptor (dictionary

/Users/jehodges/Documents/work/standards/W3C/webauthn/index-master-tr-716a169-WD-08a.txt, Top line: 7806

#dom-collectedclientdata-originReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
* 5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method
* 5.10.1. Client data used in WebAuthn signatures (dictionary CollectedClientData)
* 7.1. Registering a new credential
* 7.2. Verifying an authentication assertion

#dom-collectedclientdata-tokenbindingReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
* 5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method
* 5.10.1. Client data used in WebAuthn signatures (dictionary CollectedClientData)
* 7.1. Registering a new credential (2)
* 7.2. Verifying an authentication assertion (2)

#collectedclientdata-json-serialized-client-dataReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
* 5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method
* 5.2. Authenticator Responses (interface AuthenticatorResponse)
* 5.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse) (2)
* 5.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
* 5.10.1. Client data used in WebAuthn signatures (dictionary CollectedClientData)

#collectedclientdata-hash-of-the-serialized-client-dataReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
* 5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method
* 5.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)
* 5.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
* 6. WebAuthn Authenticator Model
* 6.2.2. The authenticatorMakeCredential operation
* 6.2.3. The authenticatorGetAssertion operation (2)
* 6.3.2. Attestation Statement Formats (2)
* 6.3.4. Generating an Attestation Object
* 7.1. Registering a new credential
* 8.2. Packed Attestation Statement Format
* 8.3. TPM Attestation Statement Format
* 8.4. Android Key Attestation Statement Format
* 8.5. Android SafetyNet Attestation Statement Format
* 8.6. FIDO U2F Attestation Statement Format

#enumdef-publickeycredentialtypeReferenced in:
* 4. Terminology
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2)
* 5.3. Parameters for Credential Generation (dictionary PublicKeyCredentialParameters)
* 5.10.2. Credential Type enumeration (enum PublicKeyCredentialType)
* 5.10.3. Credential Descriptor (dictionary

```
6885        PublicKeyCredentialDescriptor)
6886      * 6.2.1. The authenticatorMakeCredential operation (2) (3)
6887
6888    #dom-publickeycredentialtype-public-keyReferenced in:
6889      * 5.8.2. Credential Type enumeration (enum PublicKeyCredentialType)
6890
6891    #dictdef-publickeycredentialdescriptorReferenced in:
6892      * 5.1.4.1. PublicKeyCredential's
6893        [[DiscoverFromExternalSource]](origin, options,
6894        sameOriginWithAncestors) method
6895      * 5.4. Options for Credential Creation (dictionary
6896        MakePublicKeyCredentialOptions) (2)
6897      * 5.5. Options for Assertion Generation (dictionary
6898        PublicKeyCredentialRequestOptions) (2) (3)
6899      * 5.8.3. Credential Descriptor (dictionary
6900        PublicKeyCredentialDescriptor)
6901      * 6.2.1. The authenticatorMakeCredential operation
6902      * 6.2.2. The authenticatorGetAssertion operation
6903
6904    #dom-publickeycredentialdescriptor-transportsReferenced in:
6905      * 5.1.3. Create a new credential - PublicKeyCredential's
6906        [[Create]](origin, options, sameOriginWithAncestors) method (2)
6907      * 5.1.4.1. PublicKeyCredential's
6908        [[DiscoverFromExternalSource]](origin, options,
6909        sameOriginWithAncestors) method (2)
6910
6911    #dom-publickeycredentialdescriptor-typeReferenced in:
6912      * 5.1.4.1. PublicKeyCredential's
6913        [[DiscoverFromExternalSource]](origin, options,
6914        sameOriginWithAncestors) method
6915      * 5.8.3. Credential Descriptor (dictionary
6916        PublicKeyCredentialDescriptor)
6917      * 6.2.1. The authenticatorMakeCredential operation
6918      * 6.2.2. The authenticatorGetAssertion operation
6919
6920    #dom-publickeycredentialdescriptor-idReferenced in:
6921      * 5.1.4.1. PublicKeyCredential's
6922        [[DiscoverFromExternalSource]](origin, options,
6923        sameOriginWithAncestors) method (2)
6924      * 5.8.3. Credential Descriptor (dictionary
6925        PublicKeyCredentialDescriptor)
6926      * 6.2.1. The authenticatorMakeCredential operation
6927      * 6.2.2. The authenticatorGetAssertion operation
6928
6929    #enumdef-authenticatortransportReferenced in:
6930      * 5.8.3. Credential Descriptor (dictionary
6931        PublicKeyCredentialDescriptor)
6932      * 5.8.4. Authenticator Transport enumeration (enum
6933        AuthenticatorTransport)
6934
6935    #dom-authenticatortransport-usbReferenced in:
6936      * 5.8.4. Authenticator Transport enumeration (enum
6937        AuthenticatorTransport)
6938
6939    #dom-authenticatortransport-nfcReferenced in:
6940      * 5.8.4. Authenticator Transport enumeration (enum
6941        AuthenticatorTransport)
6942
6943    #dom-authenticatortransport-bleReferenced in:
6944      * 5.8.4. Authenticator Transport enumeration (enum
6945        AuthenticatorTransport)
6946
6947    #typedefdef-cosealgorithmidentifierReferenced in:
6948      * 5.1.3. Create a new credential - PublicKeyCredential's
6949        [[Create]](origin, options, sameOriginWithAncestors) method
6950      * 5.3. Parameters for Credential Generation (dictionary
6951        PublicKeyCredentialParameters)
6952      * 5.8.5. Cryptographic Algorithm Identifier (typedef
```

```
7873        PublicKeyCredentialDescriptor)
7874      * 6.2.2. The authenticatorMakeCredential operation (2) (3)
7875
7876    #dom-publickeycredentialtype-public-keyReferenced in:
7877      * 4. Terminology
7878      * 5.10.2. Credential Type enumeration (enum PublicKeyCredentialType)
7879      * 6.2.2. The authenticatorMakeCredential operation
7880
7881    #dictdef-publickeycredentialdescriptorReferenced in:
7882      * 5.1.4.1. PublicKeyCredential's
7883        [[DiscoverFromExternalSource]](origin, options,
7884        sameOriginWithAncestors) method
7885      * 5.4. Options for Credential Creation (dictionary
7886        PublicKeyCredentialCreationOptions) (2)
7887      * 5.5. Options for Assertion Generation (dictionary
7888        PublicKeyCredentialRequestOptions) (2) (3)
7889      * 5.10.3. Credential Descriptor (dictionary
7890        PublicKeyCredentialDescriptor)
7891      * 6.2.2. The authenticatorMakeCredential operation
7892      * 6.2.3. The authenticatorGetAssertion operation
7893
7894    #dom-publickeycredentialdescriptor-transportsReferenced in:
7895      * 5.1.3. Create a new credential - PublicKeyCredential's
7896        [[Create]](origin, options, sameOriginWithAncestors) method (2)
7897      * 5.1.4.1. PublicKeyCredential's
7898        [[DiscoverFromExternalSource]](origin, options,
7899        sameOriginWithAncestors) method (2)
7900
7901    #dom-publickeycredentialdescriptor-typeReferenced in:
7902      * 5.1.4.1. PublicKeyCredential's
7903        [[DiscoverFromExternalSource]](origin, options,
7904        sameOriginWithAncestors) method
7905      * 5.10.3. Credential Descriptor (dictionary
7906        PublicKeyCredentialDescriptor)
7907      * 6.2.2. The authenticatorMakeCredential operation
7908
7909    #dom-publickeycredentialdescriptor-idReferenced in:
7910      * 5.1.4.1. PublicKeyCredential's
7911        [[DiscoverFromExternalSource]](origin, options,
7912        sameOriginWithAncestors) method (2)
7913      * 5.10.3. Credential Descriptor (dictionary
7914        PublicKeyCredentialDescriptor)
7915      * 6.2.2. The authenticatorMakeCredential operation
7916      * 6.2.3. The authenticatorGetAssertion operation
7917
7918    #enumdef-authenticatortransportReferenced in:
7919      * 5.10.3. Credential Descriptor (dictionary
7920        PublicKeyCredentialDescriptor)
7921      * 5.10.4. Authenticator Transport enumeration (enum
7922        AuthenticatorTransport)
7923
7924    #dom-authenticatortransport-usbReferenced in:
7925      * 5.10.4. Authenticator Transport enumeration (enum
7926        AuthenticatorTransport)
7927
7928    #dom-authenticatortransport-nfcReferenced in:
7929      * 5.10.4. Authenticator Transport enumeration (enum
7930        AuthenticatorTransport)
7931
7932    #dom-authenticatortransport-bleReferenced in:
7933      * 5.10.4. Authenticator Transport enumeration (enum
7934        AuthenticatorTransport)
7935
7936    #typedefdef-cosealgorithmidentifierReferenced in:
7937      * 5.1.3. Create a new credential - PublicKeyCredential's
7938        [[Create]](origin, options, sameOriginWithAncestors) method
7939      * 5.3. Parameters for Credential Generation (dictionary
7940        PublicKeyCredentialParameters)
7941      * 5.10.5. Cryptographic Algorithm Identifier (typedef
```

```
6953          COSEAlgorithmIdentifier)
6954        * 6.2.1. The authenticatorMakeCredential operation
6955        * 6.3.1. Attested credential data
6956        * 8.2. Packed Attestation Statement Format
6957        * 8.3. TPM Attestation Statement Format
6958
6959      #enumdef-userverificationrequirementReferenced in:
6960        * 5.4.4. Authenticator Selection Criteria (dictionary
6961          AuthenticatorSelectionCriteria) (2)
6962        * 5.5. Options for Assertion Generation (dictionary
6963          PublicKeyCredentialRequestOptions) (2)
6964        * 5.8.6. User Verification Requirement enumeration (enum
6965          UserVerificationRequirement)
6966
6967      #dom-userverificationrequirement-requiredReferenced in:
6968        * 5.1.3. Create a new credential - PublicKeyCredential's
6969          [[Create]](origin, options, sameOriginWithAncestors) method (2)
6970        * 5.1.4.1. PublicKeyCredential's
6971          [[DiscoverFromExternalSource]](origin, options,
6972          sameOriginWithAncestors) method (2)
6973        * 5.8.6. User Verification Requirement enumeration (enum
6974          UserVerificationRequirement)
6975
6976      #dom-userverificationrequirement-preferredReferenced in:
6977        * 5.1.3. Create a new credential - PublicKeyCredential's
6978          [[Create]](origin, options, sameOriginWithAncestors) method
6979        * 5.1.4.1. PublicKeyCredential's
6980          [[DiscoverFromExternalSource]](origin, options,
6981          sameOriginWithAncestors) method
6982        * 5.8.6. User Verification Requirement enumeration (enum
6983          UserVerificationRequirement)
6984
6985      #dom-userverificationrequirement-discouragedReferenced in:
6986        * 5.1.3. Create a new credential - PublicKeyCredential's
6987          [[Create]](origin, options, sameOriginWithAncestors) method
6988        * 5.1.4.1. PublicKeyCredential's
6989          [[DiscoverFromExternalSource]](origin, options,
6990          sameOriginWithAncestors) method
6991        * 5.8.6. User Verification Requirement enumeration (enum
6992          UserVerificationRequirement)
6993
6994      #attestation-signatureReferenced in:
6995        * 4. Terminology
6996        * 6. WebAuthn Authenticator model (2) (3)
6997        * 6.3. Attestation
6998        * 7.1. Registering a new credential
6999        * 8.6. FIDO U2F Attestation Statement Format
7000
7001      #assertion-signatureReferenced in:
7002        * 6. WebAuthn Authenticator model (2)
7003        * 6.2.2. The authenticatorGetAssertion operation (2) (3)
7004
7005      #authenticator-dataReferenced in:
7006        * 5.1.4.1. PublicKeyCredential's
7007          [[DiscoverFromExternalSource]](origin, options,
7008          sameOriginWithAncestors) method
7009        * 5.2.1. Information about Public Key Credential (interface
7010          AuthenticatorAttestationResponse) (2)
7011        * 5.2.2. Web Authentication Assertion (interface
7012          AuthenticatorAssertionResponse)
7013        * 6. WebAuthn Authenticator model (2)
7014        * 6.1. Authenticator data (2) (3) (4) (5) (6) (7) (8) (9)
```



```
7942          COSEAlgorithmIdentifier)
7943        * 6.2.2. The authenticatorMakeCredential operation
7944        * 6.3.1. Attested credential data
7945        * 8.2. Packed Attestation Statement Format
7946        * 8.3. TPM Attestation Statement Format
7947
7948      #enumdef-userverificationrequirementReferenced in:
7949        * 5.4.4. Authenticator Selection Criteria (dictionary
7950          AuthenticatorSelectionCriteria) (2)
7951        * 5.5. Options for Assertion Generation (dictionary
7952          PublicKeyCredentialRequestOptions) (2)
7953        * 5.10.6. User Verification Requirement enumeration (enum
7954          UserVerificationRequirement)
7955
7956      #dom-userverificationrequirement-requiredReferenced in:
7957        * 5.1.3. Create a new credential - PublicKeyCredential's
7958          [[Create]](origin, options, sameOriginWithAncestors) method (2)
7959        * 5.1.4.1. PublicKeyCredential's
7960          [[DiscoverFromExternalSource]](origin, options,
7961          sameOriginWithAncestors) method (2)
7962        * 5.10.6. User Verification Requirement enumeration (enum
7963          UserVerificationRequirement)
7964
7965      #dom-userverificationrequirement-preferredReferenced in:
7966        * 5.1.3. Create a new credential - PublicKeyCredential's
7967          [[Create]](origin, options, sameOriginWithAncestors) method
7968        * 5.1.4.1. PublicKeyCredential's
7969          [[DiscoverFromExternalSource]](origin, options,
7970          sameOriginWithAncestors) method
7971        * 5.10.6. User Verification Requirement enumeration (enum
7972          UserVerificationRequirement)
7973
7974      #dom-userverificationrequirement-discouragedReferenced in:
7975        * 5.1.3. Create a new credential - PublicKeyCredential's
7976          [[Create]](origin, options, sameOriginWithAncestors) method
7977        * 5.1.4.1. PublicKeyCredential's
7978          [[DiscoverFromExternalSource]](origin, options,
7979          sameOriginWithAncestors) method
7980        * 5.10.6. User Verification Requirement enumeration (enum
7981          UserVerificationRequirement)
7982
7983      #authenticator-modelReferenced in:
7984        * 6. WebAuthn Authenticator Model
7985
7986      #authenticator-credentials-mapReferenced in:
7987        * 6.2.1. Lookup Credential Source by Credential ID algorithm
7988        * 6.2.2. The authenticatorMakeCredential operation
7989        * 6.2.3. The authenticatorGetAssertion operation
7990
7991      #attestation-signatureReferenced in:
7992        * 4. Terminology
7993        * 6. WebAuthn Authenticator Model (2) (3)
7994        * 6.3. Attestation
7995        * 7.1. Registering a new credential
7996        * 8.6. FIDO U2F Attestation Statement Format
7997
7998      #assertion-signatureReferenced in:
7999        * 6. WebAuthn Authenticator Model (2)
8000        * 6.2.3. The authenticatorGetAssertion operation (2) (3)
8001
8002      #authenticator-dataReferenced in:
8003        * 5.1.4.1. PublicKeyCredential's
8004          [[DiscoverFromExternalSource]](origin, options,
8005          sameOriginWithAncestors) method
8006        * 5.2.1. Information about Public Key Credential (interface
8007          AuthenticatorAttestationResponse) (2)
8008        * 5.2.2. Web Authentication Assertion (interface
8009          AuthenticatorAssertionResponse)
8010        * 6. WebAuthn Authenticator Model (2)
8011        * 6.1. Authenticator data (2) (3) (4) (5) (6) (7) (8) (9)
```

```
7015        * 6.1.1. Signature Counter Considerations (2)
7016        * 6.2.1. The authenticatorMakeCredential operation
7017        * 6.2.2. The authenticatorGetAssertion operation
7018        * 6.3. Attestation (2)
7019        * 6.3.1. Attested credential data
7020        * 6.3.2. Attestation Statement Formats (2)
7021        * 6.3.4. Generating an Attestation Object
7022        * 6.3.5.3. Attestation Certificate Hierarchy
7023        * 7.1. Registering a new credential
7024        * 8.5. Android SafetyNet Attestation Statement Format
7025        * 9.5. Authenticator extension processing
7026        * 9.6. Example Extension (2)
7027        * 10.6. User Verification Index Extension (uvi)
7028        * 10.7. Location Extension (loc)
7029        * 10.8. User Verification Method Extension (uvm)


7030
7031      #rpidhashReferenced in:
7032        * 7.2. Verifying an authentication assertion
7033
7034      #flagsReferenced in:
7035        * 5.8.6. User Verification Requirement enumeration (enum
7036        UserVerificationRequirement) (2)
7037        * 6.1. Authenticator data


7038
7039      #signcountReferenced in:
7040        * 6.1.1. Signature Counter Considerations (2)
7041        * 7.2. Verifying an authentication assertion (2) (3)
7042
7043      #attestedcredentialdataReferenced in:
7044        * 5.1.3. Create a new credential - PublicKeyCredential's
7045        [[Create]](origin, options, sameOriginWithAncestors) method
7046        * 6.1. Authenticator data (2)
7047        * 6.2.1. The authenticatorMakeCredential operation
7048        * 6.2.2. The authenticatorGetAssertion operation
7049        * 7.1. Registering a new credential (2)
7050        * 8.3. TPM Attestation Statement Format
7051        * 8.4. Android Key Attestation Statement Format
7052        * 8.6. FIDO U2F Attestation Statement Format
7053
7054      #authdataextensionsReferenced in:
7055        * 6.1. Authenticator data
7056        * 6.2.1. The authenticatorMakeCredential operation
7057        * 6.2.2. The authenticatorGetAssertion operation



7058
7059      #signature-counterReferenced in:
7060        * 6.1. Authenticator data
7061        * 6.1.1. Signature Counter Considerations (2) (3) (4) (5) (6) (7) (8)
7062        (9) (10)
7063        * 6.2.1. The authenticatorMakeCredential operation (2) (3) (4)
7064        * 6.2.2. The authenticatorGetAssertion operation (2)
7065        * 7.2. Verifying an authentication assertion (2) (3) (4) (5) (6)
7066


7067      #authenticator-sessionReferenced in:
7068        * 5.6. Abort operations with AbortSignal (2)
7069        * 6.2.1. The authenticatorMakeCredential operation
7070        * 6.2.2. The authenticatorGetAssertion operation
7071        * 6.2.3. The authenticatorCancel operation (2)
```

```
8012        * 6.1.1. Signature Counter Considerations (2)
8013        * 6.2.2. The authenticatorMakeCredential operation
8014        * 6.2.3. The authenticatorGetAssertion operation
8015        * 6.3. Attestation (2)
8016        * 6.3.1. Attested credential data
8017        * 6.3.2. Attestation Statement Formats (2)
8018        * 6.3.4. Generating an Attestation Object

8019        * 7.1. Registering a new credential
8020        * 8.5. Android SafetyNet Attestation Statement Format
8021        * 9.5. Authenticator extension processing (2)

8022        * 10.6. User Verification Index Extension (uvi)

8023        * 10.8. User Verification Method Extension (uvm)
8024        * 13.2.1. Attestation Certificate Hierarchy
8025        * 13.3. credentialId Unsigned
8026
8027      #rpidhashReferenced in:
8028        * 7.2. Verifying an authentication assertion
8029
8030      #flagsReferenced in:
8031        * 5.10.6. User Verification Requirement enumeration (enum
8032        UserVerificationRequirement) (2)
8033        * 6.1. Authenticator data
8034        * 7.1. Registering a new credential (2)
8035        * 7.2. Verifying an authentication assertion (2)
8036
8037      #signcountReferenced in:
8038        * 6.1.1. Signature Counter Considerations (2)
8039        * 7.2. Verifying an authentication assertion (2) (3)
8040
8041      #attestedcredentialdataReferenced in:
8042        * 5.1.3. Create a new credential - PublicKeyCredential's
8043        [[Create]](origin, options, sameOriginWithAncestors) method
8044        * 6.1. Authenticator data (2)
8045        * 6.2.2. The authenticatorMakeCredential operation
8046        * 6.2.3. The authenticatorGetAssertion operation
8047        * 7.1. Registering a new credential (2)
8048        * 8.3. TPM Attestation Statement Format
8049        * 8.4. Android Key Attestation Statement Format
8050        * 8.6. FIDO U2F Attestation Statement Format
8051
8052      #authdataextensionsReferenced in:
8053        * 6.1. Authenticator data
8054        * 6.2.2. The authenticatorMakeCredential operation
8055        * 6.2.3. The authenticatorGetAssertion operation
8056        * 7.1. Registering a new credential (2)
8057        * 7.2. Verifying an authentication assertion (2)
8058        * 9.5. Authenticator extension processing (2)
8059
8060      #signature-counterReferenced in:
8061        * 6.1. Authenticator data
8062        * 6.1.1. Signature Counter Considerations (2) (3) (4) (5) (6) (7) (8)
8063        (9) (10)
8064        * 6.2.2. The authenticatorMakeCredential operation (2) (3) (4)
8065        * 6.2.3. The authenticatorGetAssertion operation (2)
8066        * 7.2. Verifying an authentication assertion (2) (3) (4) (5) (6)
8067
8068      #authenticator-operationsReferenced in:
8069        * 4. Terminology
8070
8071      #authenticator-sessionReferenced in:
8072        * 5.6. Abort operations with AbortSignal (2)
8073        * 6.2.2. The authenticatorMakeCredential operation
8074        * 6.2.3. The authenticatorGetAssertion operation
8075        * 6.2.4. The authenticatorCancel operation (2)
8076
8077      #credential-id-looking-upReferenced in:
8078        * 6.2.2. The authenticatorMakeCredential operation
```

```
7072
7073      #authenticatormakecredentialReferenced in:
7074        * 4. Terminology (2) (3) (4)
7075        * 5.1.3. Create a new credential - PublicKeyCredential's
7076          [[Create]](origin, options, sameOriginWithAncestors) method (2)
7077        * 6. WebAuthn Authenticator model
7078        * 6.2.3. The authenticatorCancel operation (2)
7079        * 9. WebAuthn Extensions
7080        * 9.2. Defining extensions
7081
7082      #authenticatorgetassertionReferenced in:
7083        * 4. Terminology (2) (3)
7084        * 5.1.4.1. PublicKeyCredential's
7085          [[DiscoverFromExternalSource]](origin, options,
7086          sameOriginWithAncestors) method (2) (3) (4)
7087        * 6. WebAuthn Authenticator model
7088        * 6.1. Authenticator data
7089        * 6.1.1. Signature Counter Considerations (2) (3)
7090        * 6.2.3. The authenticatorCancel operation (2)
7091        * 9. WebAuthn Extensions
7092        * 9.2. Defining extensions
7093
7094      #authenticatorcancelReferenced in:
7095        * 5.1.3. Create a new credential - PublicKeyCredential's
7096          [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
7097          (4)
7098        * 5.1.4.1. PublicKeyCredential's
7099          [[DiscoverFromExternalSource]](origin, options,
7100          sameOriginWithAncestors) method (2) (3) (4)
7101        * 6.2.1. The authenticatorMakeCredential operation
7102        * 6.2.2. The authenticatorGetAssertion operation
7103
7104      #attestation-objectReferenced in:
7105        * 4. Terminology (2) (3)
7106        * 5. Web Authentication API
7107        * 5.2.1. Information about Public Key Credential (interface
7108          AuthenticatorAttestationResponse) (2)
7109        * 5.4. Options for Credential Creation (dictionary
7110          MakePublicKeyCredentialOptions) (2)
7111        * 6.2.1. The authenticatorMakeCredential operation (2)
7112        * 6.3. Attestation (2) (3)
7113        * 6.3.1. Attested credential data
7114        * 6.3.4. Generating an Attestation Object (2)
7115        * 7.1. Registering a new credential
7116
7117      #attestation-statementReferenced in:
7118        * 4. Terminology (2)
7119        * 5.1.3. Create a new credential - PublicKeyCredential's
7120          [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
7121        * 5.2.1. Information about Public Key Credential (interface
7122          AuthenticatorAttestationResponse) (2) (3)
7123        * 5.4.6. Attestation Conveyance Preference enumeration (enum
7124          AttestationConveyancePreference) (2) (3) (4) (5) (6) (7)
7125        * 6.3. Attestation (2) (3) (4) (5) (6) (7) (8)
7126        * 6.3.2. Attestation Statement Formats (2) (3) (4)
7127        * 7.1. Registering a new credential
7128
7129      #attestation-statement-formatReferenced in:
7130        * 5.2.1. Information about Public Key Credential (interface
7131          AuthenticatorAttestationResponse)
7132        * 5.8.4. Authenticator Transport enumeration (enum
7133          AuthenticatorTransport)
7134        * 6.2.1. The authenticatorMakeCredential operation
7135        * 6.3. Attestation (2) (3) (4) (5) (6) (7)
7136        * 6.3.2. Attestation Statement Formats (2) (3) (4)
```

```
8079        * 6.2.3. The authenticatorGetAssertion operation
8080
8081      #authenticatormakecredentialReferenced in:
8082        * 4. Terminology (2) (3) (4)
8083        * 5.1.3. Create a new credential - PublicKeyCredential's
8084          [[Create]](origin, options, sameOriginWithAncestors) method (2)
8085        * 6. WebAuthn Authenticator Model
8086        * 6.2.4. The authenticatorCancel operation (2)
8087        * 9. WebAuthn Extensions
8088        * 9.2. Defining extensions
8089        * 9.5. Authenticator extension processing
8090
8091      #authenticatorgetassertionReferenced in:
8092        * 4. Terminology (2) (3)
8093        * 5.1.4.1. PublicKeyCredential's
8094          [[DiscoverFromExternalSource]](origin, options,
8095          sameOriginWithAncestors) method (2) (3) (4)
8096        * 6. WebAuthn Authenticator Model
8097        * 6.1. Authenticator data
8098        * 6.1.1. Signature Counter Considerations (2) (3)
8099        * 6.2.4. The authenticatorCancel operation (2)
8100        * 9. WebAuthn Extensions
8101        * 9.2. Defining extensions
8102        * 9.5. Authenticator extension processing
8103        * 10.1. FIDO AppID Extension (appid)
8104
8105      #authenticatorcancelReferenced in:
8106        * 5.1.3. Create a new credential - PublicKeyCredential's
8107          [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
8108          (4) (5)
8109        * 5.1.4.1. PublicKeyCredential's
8110          [[DiscoverFromExternalSource]](origin, options,
8111          sameOriginWithAncestors) method (2) (3) (4)
8112        * 6.2.2. The authenticatorMakeCredential operation
8113        * 6.2.3. The authenticatorGetAssertion operation
8114
8115      #attestation-objectReferenced in:
8116        * 4. Terminology (2) (3)
8117        * 5. Web Authentication API
8118        * 5.2.1. Information about Public Key Credential (interface
8119          AuthenticatorAttestationResponse) (2)
8120        * 5.4. Options for Credential Creation (dictionary
8121          PublicKeyCredentialCreationOptions) (2)
8122        * 6.2.2. The authenticatorMakeCredential operation (2)
8123        * 6.3. Attestation (2) (3)
8124        * 6.3.1. Attested credential data
8125        * 6.3.4. Generating an Attestation Object (2)
8126        * 7.1. Registering a new credential
8127
8128      #attestation-statementReferenced in:
8129        * 4. Terminology (2)
8130        * 5.1.3. Create a new credential - PublicKeyCredential's
8131          [[Create]](origin, options, sameOriginWithAncestors) method (2)
8132        * 5.2.1. Information about Public Key Credential (interface
8133          AuthenticatorAttestationResponse) (2) (3)
8134        * 5.4.6. Attestation Conveyance Preference enumeration (enum
8135          AttestationConveyancePreference) (2) (3) (4) (5) (6)
8136        * 6.3. Attestation (2) (3) (4) (5) (6) (7) (8)
8137        * 6.3.2. Attestation Statement Formats (2) (3) (4)
8138        * 7.1. Registering a new credential
8139        * 8.7. None Attestation Statement Format
8140
8141      #attestation-statement-formatReferenced in:
8142        * 5.2.1. Information about Public Key Credential (interface
8143          AuthenticatorAttestationResponse)
8144        * 5.10.4. Authenticator Transport enumeration (enum
8145          AuthenticatorTransport)
8146        * 6.2.2. The authenticatorMakeCredential operation
8147        * 6.3. Attestation (2) (3) (4) (5) (6) (7)
8148        * 6.3.2. Attestation Statement Formats (2) (3) (4)
```

```
7137      * 6.3.4. Generating an Attestation Object
7138      * 7.1. Registering a new credential (2)
7139
7140    #attestation-typeReferenced in:
7141      * 5.1.3. Create a new credential - PublicKeyCredential's
7142        [[Create]](origin, options, sameOriginWithAncestors) method
7143      * 6.3. Attestation (2) (3) (4) (5) (6)
7144      * 6.3.2. Attestation Statement Formats (2)
7145
7146    #attested-credential-dataReferenced in:
7147      * 5.1.3. Create a new credential - PublicKeyCredential's
7148        [[Create]](origin, options, sameOriginWithAncestors) method
7149      * 6.1. Authenticator data (2) (3) (4) (5)
7150      * 6.2.1. The authenticatorMakeCredential operation
7151      * 6.3. Attestation (2)
7152      * 6.3.1. Attested credential data
7153      * 6.3.3. Attestation Types
7154
7155    #aaguidReferenced in:
7156      * 4. Terminology
7157      * 5.1.3. Create a new credential - PublicKeyCredential's
7158        [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
7159      * 5.4.6. Attestation Conveyance Preference enumeration (enum
7160        AttestationConveyancePreference)
7161      * 7.1. Registering a new credential
7162      * 8.2. Packed Attestation Statement Format
7163      * 8.3. TPM Attestation Statement Format
7164
7165    #credentialidlengthReferenced in:
7166      * 6.1. Authenticator data
7167
7168    #credentialidReferenced in:
7169      * 5.1.3. Create a new credential - PublicKeyCredential's
7170        [[Create]](origin, options, sameOriginWithAncestors) method
7171      * 6.1. Authenticator data
7172      * 7.1. Registering a new credential
7173
7174    #credentialpublickeyReferenced in:
7175      * 6.1. Authenticator data
          |
          |
7176      * 7.1. Registering a new credential
7177      * 8.2. Packed Attestation Statement Format
7178      * 8.3. TPM Attestation Statement Format
7179      * 8.4. Android Key Attestation Statement Format
7180
7181    #signing-procedureReferenced in:
7182      * 6.3.2. Attestation Statement Formats
7183      * 6.3.4. Generating an Attestation Object
7184
7185    #authenticator-data-for-the-attestationReferenced in:
7186      * 8.2. Packed Attestation Statement Format
7187      * 8.3. TPM Attestation Statement Format
7188      * 8.4. Android Key Attestation Statement Format (2)
7189      * 8.5. Android SafetyNet Attestation Statement Format
7190      * 8.6. FIDO U2F Attestation Statement Format
7191
7192    #verification-procedure-inputsReferenced in:
7193      * 8.2. Packed Attestation Statement Format
7194      * 8.3. TPM Attestation Statement Format
7195      * 8.4. Android Key Attestation Statement Format
7196      * 8.5. Android SafetyNet Attestation Statement Format
7197      * 8.6. FIDO U2F Attestation Statement Format
7198
7199    #authenticator-data-claimed-to-have-been-used-for-the-attestationRefere
7200    nced in:
7201      * 8.4. Android Key Attestation Statement Format
7202
7203    #attestation-trust-pathReferenced in:
7204      * 6.3.2. Attestation Statement Formats
```

```
8149      * 6.3.4. Generating an Attestation Object
8150      * 7.1. Registering a new credential (2)
8151
8152    #attestation-typeReferenced in:
8153      * 5.1.3. Create a new credential - PublicKeyCredential's
8154        [[Create]](origin, options, sameOriginWithAncestors) method
8155      * 6.3. Attestation (2) (3) (4) (5) (6)
8156      * 6.3.2. Attestation Statement Formats (2)
8157
8158    #attested-credential-dataReferenced in:
8159      * 5.1.3. Create a new credential - PublicKeyCredential's
8160        [[Create]](origin, options, sameOriginWithAncestors) method (2)
8161      * 6.1. Authenticator data (2) (3) (4) (5)
8162      * 6.2.2. The authenticatorMakeCredential operation
8163      * 6.3. Attestation (2)
8164      * 6.3.1. Attested credential data
8165      * 6.3.3. Attestation Types
8166
8167    #aaguidReferenced in:
8168      * 4. Terminology
8169      * 5.1.3. Create a new credential - PublicKeyCredential's
8170        [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
8171        (4)
8172      * 7.1. Registering a new credential
8173      * 8.2. Packed Attestation Statement Format
8174      * 8.3. TPM Attestation Statement Format
8175
8176    #credentialidlengthReferenced in:
8177      * 6.1. Authenticator data
8178
8179    #credentialidReferenced in:
8180      * 5.1.3. Create a new credential - PublicKeyCredential's
8181        [[Create]](origin, options, sameOriginWithAncestors) method
8182      * 6.1. Authenticator data
8183      * 7.1. Registering a new credential (2)
8184
8185    #credentialpublickeyReferenced in:
8186      * 6.1. Authenticator data
8187      * 6.3.1.1. Examples of credentialPublicKey Values encoded in COSE_Key
8188        format
8189      * 7.1. Registering a new credential
8190      * 8.2. Packed Attestation Statement Format
8191      * 8.3. TPM Attestation Statement Format
8192      * 8.4. Android Key Attestation Statement Format
8193
8194    #signing-procedureReferenced in:
8195      * 6.3.2. Attestation Statement Formats
8196      * 6.3.4. Generating an Attestation Object
8197
8198    #authenticator-data-for-the-attestationReferenced in:
8199      * 8.2. Packed Attestation Statement Format
8200      * 8.3. TPM Attestation Statement Format
8201      * 8.4. Android Key Attestation Statement Format (2)
8202      * 8.5. Android SafetyNet Attestation Statement Format
8203      * 8.6. FIDO U2F Attestation Statement Format
8204
8205    #verification-procedure-inputsReferenced in:
8206      * 8.2. Packed Attestation Statement Format
8207      * 8.3. TPM Attestation Statement Format
8208      * 8.4. Android Key Attestation Statement Format
8209      * 8.5. Android SafetyNet Attestation Statement Format
8210      * 8.6. FIDO U2F Attestation Statement Format
8211
8212    #authenticator-data-claimed-to-have-been-used-for-the-attestationRefere
8213    nced in:
8214      * 8.4. Android Key Attestation Statement Format
8215
8216    #attestation-trust-pathReferenced in:
8217      * 6.3.2. Attestation Statement Formats
```

```
7205      * 8.2. Packed Attestation Statement Format (2) (3)
7206      * 8.3. TPM Attestation Statement Format
7207      * 8.4. Android Key Attestation Statement Format
7208      * 8.5. Android SafetyNet Attestation Statement Format
7209      * 8.6. FIDO U2F Attestation Statement Format
7210
7211    #basic-attestationReferenced in:
7212      * 6.3.5.1. Privacy
7213      * 8.4. Android Key Attestation Statement Format
7214      * 8.5. Android SafetyNet Attestation Statement Format
7215      * 8.6. FIDO U2F Attestation Statement Format
7216
7217    #self-attestationReferenced in:
7218      * 4. Terminology (2) (3) (4)
7219
7219      * 5.4.6. Attestation Conveyance Preference enumeration (enum
7220        AttestationConveyancePreference)
7221      * 6.3. Attestation (2)
7222      * 6.3.2. Attestation Statement Formats
7223      * 6.3.3. Attestation Types
7224      * 6.3.5.2. Attestation Certificate and Attestation Certificate CA
7225        Compromise
7226      * 7.1. Registering a new credential (2) (3)
7227      * 8.2. Packed Attestation Statement Format (2)
7228      * 8.6. FIDO U2F Attestation Statement Format
7229
7230    #privacy-caReferenced in:
7231      * 5.1.3. Create a new credential - PublicKeyCredential's
7232        [[Create]](origin, options, sameOriginWithAncestors) method
7233      * 5.4.6. Attestation Conveyance Preference enumeration (enum
7234        AttestationConveyancePreference)
7235      * 6.3.5.1. Privacy
7236      * 8.3. TPM Attestation Statement Format
7237      * 8.6. FIDO U2F Attestation Statement Format
7238
7239    #elliptic-curve-based-direct-anonymous-attestationReferenced in:
7240      * 6.3.5.1. Privacy
7241
7242    #ecdaaReferenced in:
7243      * 6.3.2. Attestation Statement Formats
7244      * 6.3.3. Attestation Types
7245      * 6.3.5.2. Attestation Certificate and Attestation Certificate CA
7246        Compromise
7247      * 7.1. Registering a new credential
7248      * 8.2. Packed Attestation Statement Format (2)
7249      * 8.3. TPM Attestation Statement Format (2) (3)
7250
7251    #attestation-statement-format-identifierReferenced in:
7252      * 6.3.2. Attestation Statement Formats
7253      * 6.3.4. Generating an Attestation Object
7254
7255    #identifier-of-the-ecdaa-issuer-public-keyReferenced in:
7256      * 7.1. Registering a new credential
7257      * 8.2. Packed Attestation Statement Format
7258      * 8.3. TPM Attestation Statement Format (2)
```

```
8218      * 8.2. Packed Attestation Statement Format (2) (3)
8219      * 8.3. TPM Attestation Statement Format
8220      * 8.4. Android Key Attestation Statement Format
8221      * 8.5. Android SafetyNet Attestation Statement Format
8222      * 8.6. FIDO U2F Attestation Statement Format
8223
8224    #basic-attestationReferenced in:
8225      * 14.1. Attestation Privacy
8226
8227    #basicReferenced in:
8228      * 8.2. Packed Attestation Statement Format (2)
8229      * 8.4. Android Key Attestation Statement Format (2)
8230      * 8.5. Android SafetyNet Attestation Statement Format (2)
8231      * 8.6. FIDO U2F Attestation Statement Format (2)
8232
8233    #self-attestationReferenced in:
8234      * 4. Terminology (2) (3) (4)
8235      * 5.1.3. Create a new credential - PublicKeyCredential's
8236        [[Create]](origin, options, sameOriginWithAncestors) method
8237      * 5.4.6. Attestation Conveyance Preference enumeration (enum
8238        AttestationConveyancePreference)
8239      * 6.3. Attestation (2)
8240      * 6.3.2. Attestation Statement Formats
8241      * 6.3.3. Attestation Types
8242      * 7.1. Registering a new credential (2) (3)
8243      * 8.2. Packed Attestation Statement Format (2)
8244      * 13.2.2. Attestation Certificate and Attestation Certificate CA
8245        Compromise
8246
8247    #selfReferenced in:
8248      * 8.2. Packed Attestation Statement Format
8249
8250    #attestation-caReferenced in:
8251      * 5.4.6. Attestation Conveyance Preference enumeration (enum
8252        AttestationConveyancePreference)
8253      * 6.3.3. Attestation Types (2)
8254      * 14.1. Attestation Privacy (2)
8255
8256    #attcaReferenced in:
8257      * 8.2. Packed Attestation Statement Format
8258      * 8.3. TPM Attestation Statement Format (2)
8259      * 8.6. FIDO U2F Attestation Statement Format
8260
8261    #elliptic-curve-based-direct-anonymous-attestationReferenced in:
8262      * 14.1. Attestation Privacy
8263
8264    #ecdaaReferenced in:
8265      * 6.3.2. Attestation Statement Formats
8266      * 6.3.3. Attestation Types
8267      * 7.1. Registering a new credential
8268      * 8.2. Packed Attestation Statement Format (2) (3) (4)
8269      * 8.3. TPM Attestation Statement Format (2) (3) (4)
8270      * 13.2.2. Attestation Certificate and Attestation Certificate CA
8271        Compromise
8272
8273    #noneReferenced in:
8274      * 8.7. None Attestation Statement Format (2)
8275
8276    #attestation-statement-format-identifierReferenced in:
8277      * 6.3.2. Attestation Statement Formats
8278      * 6.3.4. Generating an Attestation Object
8279
8280    #identifier-of-the-ecdaa-issuer-public-keyReferenced in:
8281      * 7.1. Registering a new credential
8282      * 8.2. Packed Attestation Statement Format
8283      * 8.3. TPM Attestation Statement Format (2)
```

```
    #ecdaa-issuer-public-keyReferenced in:
      * 6.3.2. Attestation Statement Formats
      * 6.3.5.1. Privacy
      * 7.1. Registering a new credential
      * 8.2. Packed Attestation Statement Format (2) (3)


    #registration-extensionReferenced in:
      * 5.1.3. Create a new credential - PublicKeyCredential's
        [[Create]](origin, options, sameOriginWithAncestors) method
      * 9. WebAuthn Extensions (2) (3) (4) (5) (6)
      * 9.6. Example Extension
      * 10.2. Simple Transaction Authorization Extension (txAuthSimple)
      * 10.3. Generic Transaction Authorization Extension (txAuthGeneric)
      * 10.4. Authenticator Selection Extension (authnSel)
      * 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)


      * 11.2. WebAuthn Extension Identifier Registrations (2) (3) (4) (5)
        (6) (7)

    #authentication-extensionReferenced in:
      * 5.1.4.1. PublicKeyCredential's
        [[DiscoverFromExternalSource]](origin, options,
        sameOriginWithAncestors) method
      * 9. WebAuthn Extensions (2) (3) (4) (5) (6)
      * 9.6. Example Extension
      * 10.1. FIDO AppId Extension (appid)
      * 10.2. Simple Transaction Authorization Extension (txAuthSimple)
      * 10.3. Generic Transaction Authorization Extension (txAuthGeneric)
      * 10.6. User Verification Index Extension (uvi)
      * 10.7. Location Extension (loc)
      * 10.8. User Verification Method Extension (uvm)
      * 11.2. WebAuthn Extension Identifier Registrations (2) (3) (4) (5)
        (6)

    #client-extensionReferenced in:
      * 5.1.3. Create a new credential - PublicKeyCredential's
        [[Create]](origin, options, sameOriginWithAncestors) method
      * 5.1.4.1. PublicKeyCredential's
        [[DiscoverFromExternalSource]](origin, options,
        sameOriginWithAncestors) method
      * 5.7. Authentication Extensions (typedef AuthenticationExtensions)
      * 9. WebAuthn Extensions
      * 9.2. Defining extensions
      * 9.4. Client extension processing


    #authenticator-extensionReferenced in:
      * 5.1.3. Create a new credential - PublicKeyCredential's
        [[Create]](origin, options, sameOriginWithAncestors) method
      * 5.1.4.1. PublicKeyCredential's
        [[DiscoverFromExternalSource]](origin, options,
        sameOriginWithAncestors) method
      * 5.7. Authentication Extensions (typedef AuthenticationExtensions)
      * 9. WebAuthn Extensions (2) (3)
      * 9.2. Defining extensions (2)
      * 9.3. Extending request parameters
      * 9.5. Authenticator extension processing

    #extension-identifierReferenced in:
      * 5.1. PublicKeyCredential Interface
      * 5.1.3. Create a new credential - PublicKeyCredential's
        [[Create]](origin, options, sameOriginWithAncestors) method
      * 5.1.4.1. PublicKeyCredential's
        [[DiscoverFromExternalSource]](origin, options,
```

```
    #ecdaa-issuer-public-keyReferenced in:
      * 6.3.2. Attestation Statement Formats

      * 7.1. Registering a new credential
      * 8.2. Packed Attestation Statement Format (2) (3)
      * 14.1. Attestation Privacy

    #registration-extensionReferenced in:
      * 5.1.3. Create a new credential - PublicKeyCredential's
        [[Create]](origin, options, sameOriginWithAncestors) method
      * 9. WebAuthn Extensions (2) (3) (4) (5) (6)

      * 10.2. Simple Transaction Authorization Extension (txAuthSimple)
      * 10.3. Generic Transaction Authorization Extension (txAuthGeneric)
      * 10.4. Authenticator Selection Extension (authnSel)
      * 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)
      * 10.9. Biometric Authenticator Performance Bounds Extension
        (biometricPerfBounds)
      * 11.2. WebAuthn Extension Identifier Registrations (2) (3) (4) (5)
        (6) (7)

    #authentication-extensionReferenced in:
      * 5.1.4.1. PublicKeyCredential's
        [[DiscoverFromExternalSource]](origin, options,
        sameOriginWithAncestors) method
      * 9. WebAuthn Extensions (2) (3) (4) (5) (6)


      * 10.2. Simple Transaction Authorization Extension (txAuthSimple)
      * 10.3. Generic Transaction Authorization Extension (txAuthGeneric)
      * 10.6. User Verification Index Extension (uvi)
      * 10.7. Location Extension (loc)
      * 10.8. User Verification Method Extension (uvm)
      * 11.2. WebAuthn Extension Identifier Registrations (2) (3) (4) (5)
        (6)

    #client-extensionReferenced in:
      * 5.1.3. Create a new credential - PublicKeyCredential's
        [[Create]](origin, options, sameOriginWithAncestors) method
      * 5.1.4.1. PublicKeyCredential's
        [[DiscoverFromExternalSource]](origin, options,
        sameOriginWithAncestors) method

      * 9. WebAuthn Extensions
      * 9.2. Defining extensions
      * 9.4. Client extension processing
      * 10.1. FIDO AppID Extension (appid)

    #authenticator-extensionReferenced in:
      * 5.1.3. Create a new credential - PublicKeyCredential's
        [[Create]](origin, options, sameOriginWithAncestors) method
      * 5.1.4.1. PublicKeyCredential's
        [[DiscoverFromExternalSource]](origin, options,
        sameOriginWithAncestors) method

      * 9. WebAuthn Extensions (2) (3)
      * 9.2. Defining extensions (2)
      * 9.3. Extending request parameters
      * 9.5. Authenticator extension processing

    #extension-identifierReferenced in:
      * 5.1. PublicKeyCredential Interface
      * 5.1.3. Create a new credential - PublicKeyCredential's
        [[Create]](origin, options, sameOriginWithAncestors) method
      * 5.1.4.1. PublicKeyCredential's
        [[DiscoverFromExternalSource]](origin, options,
```

/Users/jehodges/Documents/work/standards/W3C/webauthn/index-master-tr-5e63e57-WD-07.txt, Top line: 7383

```
7383
7384      #authenticator-extension-outputReferenced in:
7385        * 6.1. Authenticator data
7386        * 9. WebAuthn Extensions (2) (3)
7387        * 9.2. Defining extensions (2) (3)
7388        * 9.4. Client extension processing
7389        * 9.5. Authenticator extension processing
7390        * 9.6. Example Extension
7391        * 10.5. Supported Extensions Extension (exts)
7392        * 10.6. User Verification Index Extension (uvi)
7393        * 10.7. Location Extension (loc)
7394        * 10.8. User Verification Method Extension (uvm)
7395
7396      #typedefdef-authenticatorselectionlistReferenced in:
7397        * 10.4. Authenticator Selection Extension (authnSel)
7398
7399      #typedefdef-aaguidReferenced in:
7400        * 10.4. Authenticator Selection Extension (authnSel)
7401
```

/Users/jehodges/Documents/work/standards/W3C/webauthn/index-master-tr-716a169-WD-08a.txt, Top line: 8414

```
8414
8415      #authenticator-extension-outputReferenced in:
8416        * 6.1. Authenticator data
8417        * 7.1. Registering a new credential
8418        * 7.2. Verifying an authentication assertion
8419        * 9. WebAuthn Extensions (2) (3) (4)
8420        * 9.2. Defining extensions (2) (3)
8421        * 9.4. Client extension processing
8422        * 9.5. Authenticator extension processing
8423        * 10.5. Supported Extensions Extension (exts)
8424        * 10.6. User Verification Index Extension (uvi)
8425        * 10.8. User Verification Method Extension (uvm)
8426
8427      #dictdef-txauthgenericargReferenced in:
8428        * 10.3. Generic Transaction Authorization Extension (txAuthGeneric)
8429
8430      #typedefdef-authenticatorselectionlistReferenced in:
8431        * 10.4. Authenticator Selection Extension (authnSel) (2)
8432
8433      #typedefdef-aaguidReferenced in:
8434        * 10.4. Authenticator Selection Extension (authnSel)
8435
8436      #typedefdef-authenticationextensionssupportedReferenced in:
8437        * 10.5. Supported Extensions Extension (exts)
8438
8439      #typedefdef-uvmentryReferenced in:
8440        * 10.8. User Verification Method Extension (uvm)
8441
8442      #typedefdef-uvmentriesReferenced in:
8443        * 10.8. User Verification Method Extension (uvm)
8444
8445      #anonymization-caReferenced in:
8446        * 5.1.3. Create a new credential - PublicKeyCredential's
8447          [[Create]](origin, options, sameOriginWithAncestors) method
8448        * 5.4.6. Attestation Conveyance Preference enumeration (enum
8449          AttestationConveyancePreference)
8450        * 14.1. Attestation Privacy (2) (3)
8451
```