THE_URL:file://localhost/Users/jehodges/documents/work/standards/W3C/WebAuthn/index-master-tr-ce7925c-WD-04.html
THE_TITLE:Web Authentication: An API for accessing Scoped Credentials
^I Jump to Table of Contents-> Pop Out Sidebar

W3C

Web Authentication: An API for accessing Scoped Credentials

W3C Working Draft, 16 February 2017

This version:
https://www.w3.org/TR/2017/WD-webauthn-20170216/

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

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

Previous Versions:

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)

Abstract

This specification defines an API enabling the creation and use of strong, attested, cryptographic scoped credentials by web applications, for the purpose of strongly authenticating users. Conceptually, one or more credentials, each scoped to a given Relying Party, are created and stored on an authenticator by the user agent in conjunction with the web application. The user agent mediates access to scoped 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

THE_URL:file://localhost/Users/jehodges/documents/work/standards/W3C/webauthn/index-master-tr-dda3e24-WD-05.html
THE_TITLE:Web Authentication: An API for accessing Public Key Credentials
^I Jump to Table of Contents-> Pop Out Sidebar

W3C

Web Authentication: An API for accessing Public Key Credentials

W3C Working Draft, 5 May 2017

This version:
https://www.w3.org/TR/2017/WD-webauthn-20170505/

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-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 strong, attested, scoped, public key-based credentials by web applications, for the purpose of strongly authenticating users. Conceptually, one or more credentials, each scoped to a given Relying Party, are created and stored on an authenticator by the user agent in conjunction with the web application. The user agent mediates access to public key credentials in order to preserve user privacy. Authenticators are responsible for ensuring that no operation is performed without user consent. Authenticators provide cryptographic proof of their properties to relying parties via attestation. This specification also describes the functional model for WebAuthn conformant authenticators, including their signature and attestation functionality.

Status of this document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at 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 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

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

Table of Contents

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

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

Table of Contents

1. Introduction

This section is not normative.

1. Introduction

This section is not normative.

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

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

Broadly, compliant authenticators protect scoped credentials, and interact with user agents to implement the Web Authentication API. Some authenticators may run on the same computing device (e.g., smart phone, tablet, desktop PC) as the user agent is running on. For instance, such an authenticator might consist of a Trusted Execution Environment (TEE) applet, a Trusted Platform Module (TPM), or a Secure Element (SE) integrated into the computing device in conjunction with some means for user 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 11 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

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 11 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 criteria for a Conforming User Agent: A User Agent MUST behave as described in this specification in order to be considered conformant. Conforming User Agents MAY implement algorithms given in this specification in any way desired, so long as the end result is indistinguishable from the result that would be obtained by the specification's algorithms. A 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]

This specification also defines a model of a conformant authenticator (see 5 WebAuthn Authenticator model). This is a set of functional and security requirements for an authenticator to be usable by a Conforming User Agent. As described in 1.1 Use Cases, an authenticator may be implemented in the operating system underlying the User Agent, or in external hardware, or a combination of both.

2.1. Dependencies

This specification relies on several other underlying specifications.

Base64url encoding

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].

DOM

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 criteria for a Conforming User Agent: A User Agent MUST behave as described in this specification in order to be considered conformant. Conforming User Agents MAY implement algorithms given in this specification in any way desired, so long as the end result is indistinguishable from the result that would be obtained by the specification's algorithms. A 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]

This specification also defines a model of a conformant authenticator (see 5 WebAuthn Authenticator model). This is a set of functional and security requirements for an authenticator to be usable by a Conforming User Agent. As described in 1.1 Use Cases, an authenticator may be implemented in the operating system underlying the User Agent, or in external hardware, or a combination of both.

2.1. Dependencies

This specification relies on several other underlying specifications, 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].

Credential Management

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

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

HTML

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

Web Cryptography API

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

Web IDL

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

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

3. Terminology

ASCII case-insensitive match

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

Assertion

See Authentication Assertion.

Attestation

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

Attestation Certificate

A X.509 Certificate for the attestation key pair used by an Authenticator to attest to its manufacture and capabilities. At registration time, the authenticator uses the attestation private key to sign the Relying Party-specific credential public key (and additional data) that it generates and returns via the authenticatorMakeCredential operation. Relying Parties use the attestation public key conveyed in the attestation certificate to verify the attestation signature. Note that in the case of self attestation, the authenticator has no distinct attestation key pair nor attestation certificate, see self attestation for details.

Authentication

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 Cryptography API

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

Web IDL

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

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

3. Terminology

Assertion

See Authentication Assertion.

Attestation

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

Attestation Certificate

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

Authentication Assertion

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

Authenticator

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

Authorization Gesture

Essentially the same as user verification.

Ceremony

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

Client

See Conforming User Agent.

Conforming User Agent

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

Credential Public Key

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

Registration

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

Authentication Assertion

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

Authenticator

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

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].

Ceremony

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

Client

See Conforming User Agent.

Conforming User Agent

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

Credential Public Key

The public key portion of 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.

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 scoped credential and associate it with the user's Relying Party account. Note that this includes employing user verification.

Relying Party

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

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

Relying Party Identifier

RP ID

An identifier for the Relying Party on whose behalf a given registration or authentication ceremony is being performed. Scoped credentials can only be used for authentication by the same entity (as identified by RP ID) that created and registered them. By default, the RP ID for a WebAuthn operation is set to the current settings object's origin. This default can be

overridden by the caller subject to certain restrictions, as specified in 4.1.1 Create a new credential - makeCredential() method and 4.1.2 Use an existing credential - getAssertion()

method.

Scoped Credential

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

User Consent

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 user verification.

Relying Party

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

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

Relying Party Identifier

RP ID

An identifier for the Relying Party on whose behalf a given registration or authentication ceremony is being performed. Public Key credentials can only be used for authentication by the same entity (as identified by RP ID) that created and registered them. By default, the RP ID for a WebAuthn operation is set to the origin specified by the relevant settings object of the CredentialsContainer object. This default can be overridden by the caller subject to certain restrictions, as specified in 4.1.3 Create a new credential - PublicKeyCredential's \[[Create]](options) method and 4.1.4 Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method.

Public Key Credential

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

Test of User Presence

TUP

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

Client-side-resident Credential Private Key

A Client-side-resident Credential Private Key is stored either on the client platform, or in some cases on the authenticator itself, e.g., in the case of a discrete first-factor roaming authenticator. Such client-side credential private key storage

User consent means the user agrees with what they are being asked, i.e., it encompasses reading and understanding prompts. User verification encompasses the means employed by the user to indicate consent.

User Verification

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

operations implies use of key material managed by the authenticator.

WebAuthn Client

See Conforming User Agent.

4. Web Authentication API

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

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).

Client-Side

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

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 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 Verified

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

WebAuthn Client

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

4. Web Authentication API

This section normatively specifies the API for creating and using public key credentials. The basic idea is that the credentials belong to the user and are managed by an authenticator, with which the Relying Party interacts through the client (consisting of the browser and underlying OS platform). Scripts can (with the user's consent) request the browser to create a new credential for future use by the Relying Party. Scripts can also request the user's permission to perform authentication operations with an existing credential. All such operations are performed in the authenticator 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 5.2 Authenticator operations, the full origin of the requester is included, and signed over, in the attestation object produced when a new credential is created as well as in all assertions produced by WebAuthn credentials.

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

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

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

```
partial interface Navigator {
    readonly attribute WebAuthentication authentication;
};
```

4.1. WebAuthentication Interface

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

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


The security properties of this API are provided by the client and the authenticator working together. The authenticator, which holds and manages credentials, ensures that all operations are scoped to a particular origin, and cannot be replayed against a different origin, by incorporating the origin in its responses. Specifically, as defined in 5.2 Authenticator operations, the full origin of the requester is included, and signed over, in the attestation object produced when a new credential is created as well as in all assertions produced by WebAuthn credentials.

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

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

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

4.1. 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]
interface PublicKeyCredential : Credential {
    readonly attribute ArrayBuffer           rawId;
    readonly attribute AuthenticatorResponse response;
    readonly attribute AuthenticationExtensions clientExtensionResults;
};
```

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.

clientExtensionResults, of type AuthenticationExtensions, readonly

};

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

4.1.1. Create a new credential - makeCredential() method

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

This method takes the following parameters:

* The accountInformation parameter specifies information about the user account for which the credential is being created. This is meant for later use by the authenticator when it needs to prompt the user to select a credential. An authenticator is only required to store one credential for any given value of accountInformation. Specifically, if an authenticator already has a credential for the specified value of id in accountInformation, and if this credential is not listed in the excludeList member of options, then after successful execution of this method:
  + Any calls to getAssertion() that do not specify allowList will not result in the older credential being offered to the user.
  + Any calls to getAssertion() that specify the older credential

This attribute contains 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.

PublicKeyCredential's interface object inherits Credential's implementation of [[CollectFromCredentialStore]](options) and [[Store]](credential), and defines its own implementation of [[DiscoverFromExternalSource]](options) and [[Create]](options).

4.1.1. CredentialRequestOptions Extension

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

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

4.1.2. CredentialCreationOptions Extension

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

```
partial dictionary CredentialCreationOptions {
   MakeCredentialOptions? publicKey;
};
```

4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method

PublicKeyCredential's interface object's implementation of the [[Create]](options) method allows scripts to call navigator.credentials.create() to request the creation of a new credential key pair and PublicKeyCredential, managed by an authenticator. The user agent will prompt the user for consent. On success, the returned promise will be resolved with a PublicKeyCredential containing an AuthenticatorAttestationResponse in the allowList may also not result in it being offered to the user.

* The cryptoParameters parameter supplies information about the desired properties of the credential to be created. The sequence is ordered from most preferred to least preferred. The platform makes a best effort to create the most preferred credential that it can.
* The attestationChallenge parameter contains a challenge intended to be used for generating the newly created credential's attestation object.
* The optional options parameter specifies additional options, as described in 4.5 Additional options for Credential Generation (dictionary ScopedCredentialOptions).

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

1. If the timeout member of options is present, check if its value lies within a reasonable range as defined by the platform and if not, correct it to the closest value lying within that range. Set adjustedTimeout to this adjusted value. If timeout was not specified, then set adjustedTimeout to a platform-specific default.
2. Let promise be a new Promise. Return promise and start a timer for adjustedTimeout milliseconds. Then asynchronously continue executing the following steps. If any fatal error is encountered in this process other than the ones enumerated below, cancel the timer, reject promise with a DOMException whose name is "UnknownError", and terminate this algorithm.
3. Set callerOrigin to the current settings object's origin. If callerOrigin is an opaque origin, reject promise with a DOMException whose name is "NotAllowedError", and terminate this algorithm. Otherwise,
   + If the rpId member of options is not present, then set rpId to callerOrigin.
   + If the rpId member of options is present, then invoke the procedure used for relaxing the same-origin restriction by setting the document.domain attribute, using rpId as the given value but without changing the current document's domain. If no errors are thrown, set rpId to the value of host as computed by this procedure. Otherwise, reject promise with a DOMException whose name is "SecurityError", and terminate this algorithm.
4. Process each element of cryptoParameters using the following steps, to produce a new sequence normalizedParameters.
   + Let current be the currently selected element of cryptoParameters.
   + If current.type does not contain a ScopedCredentialType supported by this implementation, then stop processing current and move on to the next element in cryptoParameters.
   + Let normalizedAlgorithm be the result of normalizing an algorithm [WebCryptoAPI], with alg set to current.algorithm and op set to 'generateKey'. If an error occurs during this procedure, then stop processing current and move on to the next element in cryptoParameters.
   + Add a new object of type ScopedCredentialParameters to normalizedParameters, with type set to current.type and algorithm set to normalizedAlgorithm.
5. If normalizedAlgorithm is empty and cryptoParameters was not empty, cancel the timer started in step 2, reject promise with a

object.

Note: This algorithm is synchronous; the Promise resolution/rejection is taken care of by navigator.credentials.create().

This method accepts a single argument:

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

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

1. Assert: options["publicKey"] is present.
2. Let options be the value of options["publicKey"].
3. If any of the name member of options.rp, the name member of options.user, the displayName member of options.user, or the id member of options.user are not present, return a TypeError simple exception.
4. If the timeout member of options is present, check if its value lies within a reasonable range as defined by the platform and if not, correct it to the closest value lying within that range. Set adjustedTimeout to this adjusted value. If the timeout member of options is not present, then set adjustedTimeout to a platform-specific default.
5. Let global be the PublicKeyCredential interface object's environment settings object's global object.
6. Let callerOrigin be the origin specified by this PublicKeyCredential interface object's relevant settings object. If callerOrigin is an opaque origin, return a DOMException whose name is "NotAllowedError", and terminate this algorithm.
7. If the id member of options.rp is not present, then set rpId to callerOrigin.
   Otherwise:
   1. Let effectiveDomain be the callerOrigin's effective domain.
   2. If effectiveDomain is null, then return a DOMException whose name is "SecurityError" and terminate this algorithm.
   3. 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.
   4. Set rpId to options.rp.id.
8. Let normalizedParameters be a new list whose items are pairs of PublicKeyCredentialType and a dictionary type (as returned by normalizing an algorithm).
9. For each current of options.parameters:
   1. If current.type does not contain a PublicKeyCredentialType supported by this implementation, then continue.
   2. Let normalizedAlgorithm be the result of normalizing an algorithm [WebCryptoAPI], with alg set to current.algorithm and op set to "generateKey". If an error occurs during this procedure, then continue.
   3. Append the pair of current.type and normalizedAlgorithm to normalizedParameters.
10. If normalizedParameters is empty and options.parameters is not empty, cancel the timer started in step 2, return a DOMException whose name is "NotSupportedError", and terminate this algorithm.
11. Let clientExtensions be a new map and let authenticatorExtensions

DOMException whose name is "NotSupportedError", and terminate this algorithm.
6. If the extensions member of options is present, process any extensions supported by this client platform, to produce the extension data that needs to be sent to the authenticator. If an error is encountered while processing an extension, skip that extension and do not produce any extension data for it. Call the result of this processing clientExtensions.
7. Use attestationChallenge, callerOrigin and rpId, along with the token binding key associated with callerOrigin (if any), to create a ClientData structure representing this request. Choose a hash algorithm for hashAlg and compute the clientDataJSON and its clientDataHash.
8. Initialize issuedRequests and currentlyAvailableAuthenticators to empty lists.
9. For each authenticator currently available on this platform, add the authenticator to currentlyAvailableAuthenticators unless the attachment member of options is present. In that case, let attachment be attachment, and add the authenticator to currentlyAvailableAuthenticators if its attachment modality matches attachment.
10. For each authenticator in currentlyAvailableAuthenticators: asynchronously invoke the authenticatorMakeCredential operation on that authenticator with rpId, clientDataHash, accountInformation, normalizedParameters, excludeList and clientExtensions as parameters. Add a corresponding entry to issuedRequests.
    + For each credential C in the excludeList member of options that has a non-empty transports list, optionally use only the specified transports to test for the existence of C.
11. While issuedRequests is not empty, perform the following actions

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:

    challenge
        The base64url encoding of options.challenge

    origin
        The unicode serialization of rpId

    hashAlg
        The recognized algorithm name of the hash algorithm selected by the client for generating the hash of the serialized client data

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

    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. Let currentlyAvailableAuthenticators be a new ordered set consisting of all authenticators available on this platform.
17. Let selectedAuthenticators be a new ordered set.
18. If currentlyAvailableAuthenticators is empty, return a DOMException whose name is "NotFoundError", and terminate this algorithm.
19. If options.authenticatorSelection is present, iterate through currentlyAvailableAuthenticators and do the following for each authenticator:
    1. If attachment is present and its value is not equal to authenticator's attachment modality, continue.
    2. If requireResidentKey is set to true and the authenticator is not capable of storing a Client-Side-Resident Credential Private Key, continue.
    3. Append authenticator to selectedAuthenticators.
20. If selectedAuthenticators is empty, return a DOMException whose name is "ConstraintError", and terminate this algoritm.
21. Let issuedRequests be a new ordered set.
22. For each authenticator in currentlyAvailableAuthenticators:
    1. Let excludeList be a new list.
    2. For each credential C in options.excludeList:
        1. If C.transports is not empty, and authenticator is connected over a transport not mentioned in C.transports, the client MAY continue.
        2. Otherwise, Append C to excludeList.
    3. In parallel, invoke the authenticatorMakeCredential operation on authenticator with rpId, clientDataHash, options.rp, options.user, normalizedParameters, excludeList, and authenticatorExtensions as parameters.

depending upon the adjustedTimeout timer and responses from the
authenticators:
+ If the adjustedTimeout timer expires, then for each entry in
issuedRequests invoke the authenticatorCancel operation on
that authenticator and remove its entry from the list.
+ If any authenticator returns a status indicating that the user
cancelled the operation, delete that authenticator's entry
from issuedRequests. For each remaining entry in
issuedRequests invoke the authenticatorCancel operation on
that authenticator and remove its entry from the list.
+ If any authenticator returns an error status, delete the
corresponding entry from issuedRequests.
+ If any authenticator indicates success:
o Remove this authenticator's entry from issuedRequests.
o Create a new ScopedCredentialInfo object named value and
populate its fields with the values returned from the
authenticator as well as the clientDataJSON computed
earlier.
o For each remaining entry in issuedRequests invoke the
authenticatorCancel operation on that authenticator and
remove its entry from the list.
o Resolve promise with value and terminate this algorithm.

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

During the above process, the user agent SHOULD show some UI to the

4. Append authenticator to issuedRequests.

23. Start a timer for adjustedTimeout milliseconds. Then execute the
following steps in parallel. The task source for these tasks is the
dom manipulation task source.
24. While issuedRequests is not empty, perform the following actions
depending upon the adjustedTimeout timer and responses from the
authenticators:

If the adjustedTimeout timer expires,
For each authenticator in issuedRequests invoke the
authenticatorCancel operation on authenticator and remove
authenticator from issuedRequests.

If any authenticator returns a status indicating that the user
cancelled the operation,

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

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

If any authenticator indicates success,

1. Remove authenticator from issuedRequests.
2. Let attestationObject be a new ArrayBuffer, created using
global's %ArrayBuffer%, containing the bytes of the value
returned from the successful authenticatorMakeCredential
operation (which is attObj, as defined in 5.3.4
Generating an Attestation Object).
3. Let id be attestationObject.authData.attestation
data.credential ID (see 5.3.1 Attestation data and 5.1
Authenticator data).
4. Let value 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 clientDataJSON.

attestationObject
attestationObject

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

5. For each remaining authenticator in issuedRequests invoke
the authenticatorCancel operation on authenticator and
remove it from issuedRequests.
6. Return value and terminate this algorithm.

25. 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.

4.1.2. Use an existing credential - getAssertion() method

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

This method takes the following parameters:
 * The assertionChallenge parameter contains a challenge that the selected authenticator is expected to sign to produce the assertion.
 * The optional options parameter specifies additional options, as described in 4.7 Additional options for Assertion Generation (dictionary AssertionOptions).

When this method is invoked, the user agent MUST execute the following algorithm:
1. If the timeout member of options is present, check if its value lies within a reasonable range as defined by the platform and if not, correct it to the closest value lying within that range. Set adjustedTimeout to this adjusted value. If timeout was not specified, then set adjustedTimeout to a platform-specific default.
2. Let promise be a new Promise. Return promise and start a timer for adjustedTimeout milliseconds. Then asynchronously continue executing the following steps. If any fatal error is encountered in this process other than the ones enumerated below, cancel the timer, reject promise with a DOMException whose name is "UnknownError", and terminate this algorithm.
3. Set callerOrigin to the current settings object's origin. If callerOrigin is an opaque origin, reject promise with a DOMException whose name is "NotAllowedError", and terminate this algorithm. Otherwise,
   + If the rpId member of options is not present, then set rpId to callerOrigin.
   + If the rpId member of options is present, then invoke the procedure used for relaxing the same-origin restriction by setting the document.domain attribute, using rpId as the given value but without changing the current document's domain. If no errors are thrown, set rpId to the value of host as computed by this procedure. Otherwise, reject promise with a DOMException whose name is "SecurityError", and terminate this algorithm.
4. If the extensions member of options is present, process any extensions supported by this client platform, to produce the extension data that needs to be sent to the authenticator. If an error is encountered while processing an extension, skip that extension and do not produce any extension data for it. Call the result of this processing clientExtensions.
5. Use assertionChallenge, callerOrigin and rpId, along with the token binding key associated with callerOrigin (if any), to create a ClientData structure representing this request. Choose a hash algorithm for hashAlg and compute the clientDataJSON and clientDataHash.
6. Initialize issuedRequests to an empty list.
7. For each authenticator currently available on this platform,

user to guide them in the process of selecting and authorizing an authenticator.

4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method

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

Note: This algorithm is synchronous; the Promise resolution/rejection is taken care of by navigator.credentials.get().
This method takes the following parameters:

options
    A CredentialRequestOptions object, containing a challenge that the selected authenticator is expected to sign to produce the assertion, and additional options as described in 4.6 Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)

When this method is invoked, the user agent MUST execute the following algorithm:
1. Let publicKeyOptions be the value of options publicKey member.
2. If the timeout member of publicKeyOptions is present, check if its value lies within a reasonable range as defined by the platform and if not, correct it to the closest value lying within that range. Set adjustedTimeout to this adjusted value. If the timeout member of publicKeyOptions is not present, then set adjustedTimeout to a platform-specific default.
3. Let global be the PublicKeyCredential's relevant settings object's environment settings object's global object.
4. Let callerOrigin be the origin of this CredentialsContainer object's relevant settings object. If callerOrigin is an opaque origin, return DOMException whose name is "NotAllowedError", and terminate this algorithm.
5. If the rpId member of publicKeyOptions is not present, then set rpId to callerOrigin. Otherwise:
   1. Let effectiveDomain be the callerOrigin's effective domain.
   2. If effectiveDomain is null, then return a DOMException whose name is "SecurityError" and terminate this algorithm.
   3. If 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.
   4. Set rpId to the rpId.
6. Let clientExtensions be a new map and let authenticatorExtensions be a new map.
7. If the extensions member of publicKeyOptions is present, then for each extensionId -> clientExtensionInput of publicKeyOptions.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.
8. Let collectedClientData be a new CollectedClientData instance whose fields are:

    challenge

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

8. While issuedRequests is not empty, perform the following actions
depending upon the adjustedTimeout timer and responses from the
authenticators:
+ If the timer for adjustedTimeout expires, then for each entry
in issuedRequests invoke the authenticatorCancel operation on
that authenticator and remove its entry from the list.
+ If any authenticator returns a status indicating that the user
cancelled the operation, delete that authenticator's entry
from issuedRequests. For each remaining entry in
issuedRequests invoke the authenticatorCancel operation on
that authenticator, and remove its entry from the list.
+ If any authenticator returns an error status, delete the
corresponding entry from issuedRequests.
+ If any authenticator returns success:
o Remove this authenticator's entry from issuedRequests.
o Create a new AuthenticationAssertion object named value
and populate its fields with the values returned from the

The base64url encoding of publicKeyOptions.challenge

origin
The unicode serialization of rpId

hashAlg
The recognized algorithm name of the hash algorithm
selected by the client for generating the hash of the
serialized client data

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

clientExtensions
clientExtensions

authenticatorExtensions
authenticatorExtensions

9. Let clientDataJSON be the JSON-serialized client data constructed
from collectedClientData.
10. Let clientDataHash be the hash of the serialized client data
represented by clientDataJSON.
11. Let issuedRequests be a new ordered set.
12. If there are no authenticators currently available on this
platform, return a DOMException whose name is "NotFoundError", and
terminate this algorithm.
13. For each authenticator currently available on this platform,
perform the following steps:
1. Let credentialList be a new list.
2. If publicKeyOptions.allowList is not empty, execute a
platform-specific procedure to determine which, if any,
credentials in publicKeyOptions.allowList are present on this
authenticator by matching with publicKeyOptions.allowList.id
and publicKeyOptions.allowList.type, and set credentialList to
this filtered list.
3. If credentialList is empty then continue.
4. In parallel, for each credential C in credentialList:
1. If C.transports is not empty, the client SHOULD select
one transport from transports. Then, using transport,
invoke the authenticatorGetAssertion operation on
authenticator, with rpId, clientDataHash, credentialList,
and authenticatorExtensions as parameters.
2. Otherwise, using local configuration knowledge of the
appropriate transport to use with authenticator, invoke
the authenticatorGetAssertion operation on authenticator
with rpId, clientDataHash, credentialList, and
clientExtensions as parameters.
5. Append authenticator to issuedRequests.
14. Start a timer for adjustedTimeout milliseconds. Then execute the
following steps in parallel. The task source for these tasks is the
dom manipulation task source.
15. While issuedRequests is not empty, perform the following actions
depending upon the adjustedTimeout timer and responses from the
authenticators:

If the adjustedTimeout timer expires,
For each authenticator in issuedRequests invoke the
authenticatorCancel operation on authenticator and remove
authenticator from issuedRequests.

If any authenticator returns a status indicating that the user
cancelled the operation,

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

authenticator as well as the clientDataJSON computed earlier.
o For each remaining entry in issuedRequests invoke the authenticatorCancel operation on that authenticator and remove its entry from the list.
o Resolve promise with value and terminate this algorithm.
9. Reject promise with a DOMException whose name is "NotAllowedError", and terminate this algorithm.

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

4.2. Information about Scoped Credential (interface ScopedCredentialInfo)

```
[SecureContext]
interface ScopedCredentialInfo {
    readonly    attribute ArrayBuffer   clientDataJSON;
    readonly    attribute ArrayBuffer   attestationObject;
};
```

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

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

If any authenticator indicates success,

1. Remove authenticator from issuedRequests.
2. Let value be a new PublicKeyCredential associated with global whose fields are:

[[identifier]]
A new ArrayBuffer, created using global's %ArrayBuffer%, containing the bytes of the credential ID returned from the successful authenticatorGetAssertion operation, as defined in [#op-get-assertion]].

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

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

authenticatorData
A new ArrayBuffer, created using global's %ArrayBuffer%, containing the bytes of the returned authenticatorData

signature
A new ArrayBuffer, created using global's %ArrayBuffer%, containing the bytes of the returned signature

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

3. For each remaining authenticator in issuedRequests invoke the authenticatorCancel operation on authenticator and remove it from issuedRequests.
4. Return value and terminate this algorithm.

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

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

4.2. Authenticator Responses (interface AuthenticatorResponse)

Authenticators respond to relying party requests by returning an object derived from the AuthenticatorResponse interface:

```
[SecureContext]
interface AuthenticatorResponse {
    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().

The clientDataJSON attribute contains the clientDataJSON (see 5.3 Credential Attestation) passed to the authenticator by the client in order to generate this credential. The exact JSON serialization must be preserved as a cryptographic hash (clientDataHash) has been computed over it.

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

4.3. User Account Information (dictionary Account)

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

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

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

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

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

The name member contains a detailed name for the account, such as

4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

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

```
[SecureContext]
interface AuthenticatorAttestationResponse : AuthenticatorResponse {
  readonly attribute ArrayBuffer attestationObject;
};
```

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

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

4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)

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

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

clientDataJSON

"john.p.smith@example.com".

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

4.4. Parameters for Credential Generation (dictionary ScopedCredentialParameters)

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

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

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

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

4.5. Additional options for Credential Generation (dictionary ScopedCredentialOptions)

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

This dictionary is used to supply additional options when creating a new credential. All these parameters are optional.
* The timeout parameter specifies a time, in milliseconds, that the caller is willing to wait for the call to complete. This is treated

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

authenticatorData, of type ArrayBuffer, readonly
This attribute contains the authenticator data returned by the authenticator. See 5.1 Authenticator data.

signature, of type ArrayBuffer, readonly
This attribute contains the raw signature returned from the authenticator. See 5.2.2 The authenticatorGetAssertion operation.

4.3. Parameters for Credential Generation (dictionary PublicKeyCredentialParameters)

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

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

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

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

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

```
dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
  DOMString displayName;
};
```

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

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

4.5. Options for Credential Creation (dictionary MakeCredentialOptions)

```
dictionary MakeCredentialOptions {
  required PublicKeyCredentialEntity rp;
  required PublicKeyCredentialUserEntity user;

  required BufferSource                          challenge;
  required sequence<PublicKeyCredentialParameters> parameters;

  unsigned long                       timeout;
  sequence<PublicKeyCredentialDescriptor> excludeList;
  AuthenticatorSelectionCriteria      authenticatorSelection;
  AuthenticationExtensions            extensions;
};
```

rp, of type PublicKeyCredentialEntity
This member contains data about the relying party responsible for the request.

as a hint, and may be overridden by the platform.
* The rpId parameter explicitly specifies the RP ID that the credential should be associated with. If it is omitted, the RP ID will be set to the current settings object's origin.
* The excludeList parameter is intended for use by Relying Parties that wish to limit the creation of multiple credentials for the same account on a single authenticator. The platform is requested to return an error if the new credential would be created on an authenticator that also contains one of the credentials enumerated in this parameter.
* The extensions parameter contains additional parameters requesting additional processing by the client and authenticator. For example, the caller may request that only authenticators with certain capabilities be used to create the credential, or that particular information be returned in the attestation object. The caller may also specify an additional message that they would like the authenticator to display to the user. Extensions are defined in 8 WebAuthn Extensions.
* The attachment parameter contains authenticator attachment descriptions, which are used as an additional constraint on which authenticators are eligible to participate in a 4.1.1 Create a new credential - makeCredential() method or 4.1.2 Use an existing

Its value's name member is required, and 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 this identifier is not explicitly set, it will default to the ASCII serialization of the CredentialsContainer object's relevant settings object's origin.

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 is required, and contains a name for the user account (e.g., "john.p.smith@example.com" or "+14255551234").

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

  Its value's id member is required, and contains an identifier for the account, specified by the relying party. This 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 id.

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

parameters, 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.

excludeList, of type sequence<PublicKeyCredentialDescriptor>
: 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() or get() operation.

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 capabilities be used to create the credential, or that particular information be returned in the attestation object. Some extensions are defined in 8 WebAuthn Extensions; consult the IANA "WebAuthn Extension Identifier" registry established by [WebAuthn-Registries] for an up-to-date list of registered WebAuthn Extensions.

credential - getAssertion() method operation. See 4.5.1 Credential Attachment enumeration (enum Attachment) for a description of the attachment values and their meanings.

4.5.1. Credential Attachment enumeration (enum Attachment)

```
enum Attachment {
    "platform",
    "cross-platform"
};
```

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

4.5.1. Entity Description

The PublicKeyCredentialEntity dictionary describes a user account or a relying party with which a credential is associated.

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

id, of type DOMString
: A unique identifier for the entity. This will be the ASCII serialization of an origin for a relying party, and an arbitrary string specified by the relying party for user accounts.

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.

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.

4.5.2. Authenticator Selection Criteria

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

```
dictionary AuthenticatorSelectionCriteria {
    Attachment    attachment;
    boolean       requireResidentKey = false;
};
```

attachment, of type Attachment
: If this memeber is present, eligible authenticators are filtered to only authenticators attached with the specified 4.5.3 Credential Attachment enumeration (enum Attachment).

requireResidentKey, of type boolean, defaulting to false
: This member describes the Relying Parties' requirements regarding availability of the Client-side-resident Credential Private Key. If the parameter is set to true, the authenticator MUST create a Client-side-resident Credential Private Key when creating a public key credential.

4.5.3. Credential Attachment enumeration (enum Attachment)

```
enum Attachment {
    "platform",
    "cross-platform"
};
```

Clients may communicate with authenticators using a variety of mechanisms. For example, a client may use a platform-specific API to communicate with an authenticator which is physically bound to a platform. On the other hand, a client may use a variety of standardized cross-platform transport protocols such as Bluetooth (see 4.8.4 Credential Transport enumeration (enum ExternalTransport)) to discover and communicate with cross-platform attached authenticators. Therefore, we use Attachment to describe an authenticator's attachment modality. We define authenticators that are part of the client's platform as having a platform attachment, and refer to them as platform authenticators. While those that are reachable via cross-platform transport protocols are defined as having cross-platform attachment, and refer to them as roaming authenticators.

* platform attachment - the respective authenticator is attached using 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.

4.6. Web Authentication Assertion (interface AuthenticationAssertion)

```
[SecureContext]
interface AuthenticationAssertion {
    readonly attribute ScopedCredential  credential;
    readonly attribute ArrayBuffer       clientDataJSON;
    readonly attribute ArrayBuffer       authenticatorData;
    readonly attribute ArrayBuffer       signature;
};
```

Scoped credentials produce a cryptographic signature that provides proof of possession of a private key as well as evidence of user consent to a specific transaction. The structure of these signatures is defined as follows.

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

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

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

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

4.7. Additional options for Assertion Generation (dictionary AssertionOptions)

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

This dictionary is used to supply additional options when generating an assertion. All these parameters are optional.
* The optional timeout parameter specifies a time, in milliseconds, that the caller is willing to wait for the call to complete. This is treated as a hint, and may be overridden by the platform.
* The optional rpId parameter specifies the rpId claimed by the caller. If it is omitted, it will be assumed to be equal to the current settings object's origin.
* The optional allowList member contains a list of credentials

* 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.

4.6. 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> allowList = [];
    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.

acceptable to the caller, in order of the caller's preference.
* The optional extensions parameter contains additional parameters requesting additional processing by the client and authenticator. For example, if transaction confirmation is sought from the user, then the prompt string would be included in an extension. Extensions are defined in a companion specification.

4.8. Authentication Assertion Extensions (dictionary AuthenticationExtensions)

```
dictionary AuthenticationExtensions {
};
```

This is a dictionary containing zero or more extensions as defined in 8 WebAuthn Extensions. An extension is an additional parameter that can be passed to the getAssertion() method and triggers some additional processing by the client platform and/or the authenticator.

If the caller wishes to pass extensions to the platform, it MUST do so by adding one entry per extension to this dictionary with the extension identifier as the key, and the extension's value as the value (see 8 WebAuthn Extensions for details).

4.9. Supporting Data Structures

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

4.9.1. Client data used in WebAuthn signatures (dictionary ClientData)

The client data represents the contextual bindings of both the Relying Party and the client platform. It is a key-value mapping with string-valued keys. Values may be any type that has a valid encoding in JSON. Its structure is defined by the following Web IDL.

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

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

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

The hashAlg member specifies the hash algorithm used to compute

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 ASCII serialization of the CredentialsContainer object's relevant settings object's origin.

allowList, of type sequence<PublicKeyCredentialDescriptor>, defaulting to None
This optional member contains a list of PublicKeyCredentialDescriptor object 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 line).

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.

4.7. Authentication Extensions (typedef AuthenticationExtensions)

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

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

4.8. Supporting Data Structures

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

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

The client data represents the contextual bindings of both the Relying Party and the client platform. It is a key-value mapping with string-valued keys. Values may be any type that has a valid encoding in JSON. Its structure is defined by the following Web IDL.

```
dictionary CollectedClientData {
    required DOMString           challenge;
    required DOMString           origin;
    required DOMString           hashAlg;
    DOMString                    tokenBinding;
    AuthenticationExtensions     clientExtensions;
    AuthenticationExtensions     authenticatorExtensions;
};
```

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

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 hashAlg member is a recognized algorithm name that supports the clientDataHash. Use "S256" for SHA-256, "S384" for SHA384, "S512" for SHA512, and "SM3" for SM3 (see 10 IANA Considerations). This algorithm is chosen by the client at its sole discretion.

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

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

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

clientDataJSON

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

clientDataHash

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

4.9.2. Credential Type enumeration (enum ScopedCredentialType)

```
enum ScopedCredentialType {
  "ScopedCred"
};
```

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

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

4.9.3. Unique Identifier for Credential (interface ScopedCredential)

```
[SecureContext]
interface ScopedCredential {
  readonly attribute ScopedCredentialType type;
  readonly attribute ArrayBuffer          id;
};
```

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

The type attribute contains a value of type ScopedCredentialType, indicating the specification and version that this credential conforms to.

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

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

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

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

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

4.8.2. Credential Type enumeration (enum PublicKeyCredentialType)

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

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

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

consist of the key material wrapped with a key that is burned into the authenticator.

4.9.4. Credential Descriptor (dictionary ScopedCredentialDescriptor)

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

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

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

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

4.9.5. Credential Transport enumeration (enum ExternalTransport)

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

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

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

4.9.6. Cryptographic Algorithm Identifier (type AlgorithmIdentifier)

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

5. WebAuthn Authenticator model

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

Client platforms may implement and expose this abstract model in any way desired. However, the behavior of the client's Web Authentication API implementation, when operating on the authenticators supported by that platform, MUST be indistinguishable from the behavior specified in 4 Web Authentication API.

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

4.8.3. Credential Descriptor (dictionary PublicKeyCredentialDescriptor)

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

This dictionary contains the attributes that are specified by a caller when referring to a credential as an input parameter to the 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.

4.8.4. Credential Transport enumeration (enum ExternalTransport)

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

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

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

4.8.5. Cryptographic Algorithm Identifier (type AlgorithmIdentifier)

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

5. WebAuthn Authenticator model

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

Client platforms may implement and expose this abstract model in any way desired. However, the behavior of the client's Web Authentication API implementation, when operating on the authenticators supported by that platform, MUST be indistinguishable from the behavior specified in 4 Web Authentication API.

For authenticators, this model defines the logical operations that they must support, and the data formats that they expose to the client and the Relying Party. However, it does not define the details of how authenticators communicate with the client platform, unless they are required for interoperability with Relying Parties. For instance, 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 scoped credentials. Each scoped credential has an identifier which is unique (or extremely unlikely to be duplicated) among all scoped credentials. Each credential is also associated with a Relying Party, whose identity is represented by a Relying Party Identifier (RP ID).

Each authenticator has an AAGUID, which is a 128-bit identifier that indicates the type (e.g. make and model) of the authenticator. The AAGUID MUST be chosen by the manufacturer to be identical across all substantially identical authenticators made by that manufacturer, and different (with probability 1-2^-128 or greater) 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 ClientData and sends only the result to the authenticator. The authenticator signs over the combination of this clientDataHash, and its own authenticator data.

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

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

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

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

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

5.1. Authenticator data

The authenticator data structure, authenticatorData, encodes contextual bindings made by the authenticator. These bindings are controlled by the authenticator itself, and 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 ClientData. 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 encoding of authenticator data is a byte array of 37 bytes or more, as follows.

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

4 Signature counter (signCount), 32-bit unsigned big-endian integer.
variable (if present) Attestation data (if present). See 5.3.1 Attestation data for details. Its length depends on the length of the credential public key and credential ID being attested.
variable (if present) Extension-defined authenticator data. This is a CBOR [RFC7049] map with extension identifiers as keys, and extension authenticator data values as values. See 8 WebAuthn Extensions for details.

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

The TUP flag SHALL be set if and only if the authenticator detected a user through an authenticator specific gesture. The RFU bits in the flags byte SHALL be set to zero.

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

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

5.1. Authenticator data

The authenticator data structure 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.

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

4 Signature counter (signCount), 32-bit unsigned big-endian integer.
variable (if present) attestation data (if present). See 5.3.1 Attestation data for details. Its length depends on the length of the credential public key and credential ID being attested.
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 8 WebAuthn Extensions for details.

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

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

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

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

The figure below shows a visual representation of the authenticator data structure.
[fido-signature-formats-figure1.svg] authenticatorData layout.

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

5.2. Authenticator operations

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

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

5.2.1. The authenticatorMakeCredential operation

This operation must be invoked in an authenticator session which has no other operations in progress. It takes the following input parameters:
* The caller's RP ID, as determined by the user agent and the client.
* The clientDataHash, which is the hash of the serialized ClientData and is provided by the client.
* The Account information provided by the Relying Party.
* The ScopedCredentialType and cryptographic parameters requested by the Relying Party, with the cryptographic algorithms normalized as per the procedure in Web Cryptography API algorithm-normalization-normalize-an-algorithm.
* A list of ScopedCredential objects provided by the Relying Party with the intention that, if any of these are known to the authenticator, it should not create a new credential.
* Extension data created by the client based on the extensions requested by the Relying Party.

When this operation is invoked, the authenticator must perform the following procedure:
* Check if all the supplied parameters are syntactically well-formed and of the correct length. If not, return an error code equivalent to UnknownError and terminate the operation.
* Check if at least one of the specified combinations of ScopedCredentialType and cryptographic parameters is supported. If not, return an error code equivalent to NotSupportedError and terminate the operation.
* Check if a credential matching any of the supplied ScopedCredential identifiers is present on this authenticator. If so, return an error code equivalent to NotAllowedError and terminate the operation.
* Prompt the user for consent to create a new credential. The prompt for obtaining this consent is shown by the authenticator if it has its own output capability, or by the user agent otherwise. If the

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

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

The figure below shows a visual representation of the authenticator data structure.
[fido-signature-formats-figure1.html] Authenticator data layout.

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

5.2. Authenticator operations

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

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

5.2.1. The authenticatorMakeCredential operation

This operation must be invoked in an authenticator session which has no other operations in progress. It takes the following input parameters:
* The caller's RP ID, as determined by the user agent and the client.
* The hash of the serialized client data, provided by the client.
* The relying party's PublicKeyCredentialEntity.
* The user account's PublicKeyCredentialEntity.
* The PublicKeyCredentialType and cryptographic parameters requested by the Relying Party, with the cryptographic algorithms normalized as per the procedure in Web Cryptography API algorithm-normalization-normalize-an-algorithm.
* A list of PublicKeyCredential objects provided by the Relying Party with the intention that, if any of these are known to the authenticator, it should not create a new credential.
* Extension data created by the client based on the extensions requested by the Relying Party.
* The requireResidentKey parameter of the options.authenticatorSelection dictionary.

When this operation is invoked, the authenticator must perform the following procedure:
* Check if all the supplied parameters are syntactically well-formed and of the correct length. If not, return an error code equivalent to UnknownError and terminate the operation.
* Check if at least one of the specified combinations of PublicKeyCredentialType and cryptographic parameters is supported. If not, return an error code equivalent to NotSupportedError and terminate the operation.
* Check if a credential matching any of the supplied PublicKeyCredential identifiers is present on this authenticator. If so, return an error code equivalent to NotAllowedError and terminate the operation.
* If the requireResidentKey flag is set to true and the authenticator cannot store a Client-side-resident Credential Private Key, return an error code equivalent to ConstraintError and terminate the operation.
* Prompt the user for consent to create a new credential. The prompt for obtaining this consent is shown by the authenticator if it has its own output capability, or by the user agent otherwise. If the

user denies consent, return an error code equivalent to
NotAllowedError and terminate the operation.
* Once user consent has been obtained, generate a new credential
object:
+ Generate a set of cryptographic keys using the most preferred
combination of ScopedCredentialType and cryptographic
parameters supported by this authenticator.
+ Generate an identifier for this credential, such that this
identifier is globally unique with high probability across all
credentials with the same type across all authenticators.
+ Associate the credential with the specified RP ID and the
user's account identifier id.
+ Delete any older credentials with the same RP ID and id that
are stored locally in the authenticator.
* If any error occurred while creating the new credential object,
return an error code equivalent to UnknownError and terminate the
operation.
* Process all the supported extensions requested by the client, and
generate an authenticatorData structure with attestation data as
specified in 5.1 Authenticator data. Use this authenticatorData
and the clientDataHash received from the client to create an
attestation object for the new credential using the procedure
specified in 5.3.4 Generating an Attestation Object. For more
details on attestation, see 5.3 Credential Attestation.

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

5.2.2. The authenticatorGetAssertion operation

This operation must be invoked in an authenticator session which has no
other operations in progress. It takes the following input parameters:
* The caller's RP ID, as determined by the user agent and the client.
* The clientDataHash, which is the hash of the serialized ClientData
and is provided by the client.
* A list of credentials acceptable to the Relying Party (possibly
filtered by the client).
* Extension data created by the client based on the extensions
requested by the Relying Party.

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

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

user denies consent, return an error code equivalent to
NotAllowedError and terminate the operation.
* Once user consent has been obtained, generate a new credential
object:
+ Generate a set of cryptographic keys using the most preferred
combination of PublicKeyCredentialType and cryptographic
parameters supported by this authenticator.
+ Generate an identifier for this credential, such that this
identifier is globally unique with high probability across all
credentials with the same type across all authenticators.
+ Associate the credential with the specified RP ID and the
user's account identifier user.id.
+ Delete any older credentials with the same RP ID and user.id
that are stored locally in the authenticator.
* If any error occurred while creating the new credential object,
return an error code equivalent to UnknownError and terminate the
operation.
* Process all the supported extensions requested by the client, and
generate the authenticator data with attestation data as specified
in 5.1 Authenticator data. Use this authenticator data and the
hash of the serialized client data to create an attestation object
for the new credential using the procedure specified in 5.3.4
Generating an Attestation Object. For more details on attestation,
see 5.3 Credential Attestation.

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

5.2.2. The authenticatorGetAssertion operation

This operation must be invoked in an authenticator session which has no
other operations in progress. It takes the following input parameters:
* The caller's RP ID, as determined by the user agent and the client.
* The hash of the serialized client data, provided by the client.

* A list of credentials acceptable to the Relying Party (possibly
filtered by the client).
* Extension data created by the client based on the extensions
requested by the Relying Party.

When this method is invoked, the authenticator must perform the
following procedure:
* Check if all the supplied parameters are syntactically well-formed
and of the correct length. If not, return an error code equivalent
to UnknownError and terminate the operation.
* If a list of credentials was supplied by the client, filter it by
removing those credentials that are not present on this
authenticator. If no list was supplied, create a list with all
credentials stored for the caller's RP ID (as determined by an
exact match of the RP ID).
* If the previous step resulted in an empty list, return an error
code equivalent to NotAllowedError and terminate the operation.
* Prompt the user to select a credential from among the above list.
Obtain user consent for using this credential. The prompt for
obtaining this consent may be shown by the authenticator if it has
its own output capability, or by the user agent otherwise.
* Process all the supported extensions requested by the client, and
generate the authenticator data without attestation data as
specified in 5.1 Authenticator data. Concatenate this
authenticator data with the hash of the serialized client data to
generate an assertion signature using the private key of the
selected credential as shown 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.
* If any error occurred while generating the assertion signature,
return an error code equivalent to UnknownError and terminate the
operation.

[fido-signature-formats-figure2.svg] Generating a signature on the authenticator.

On successful completion, the authenticator returns to the user agent:
 * The identifier of the credential used to generate the signature.
 * The authenticatorData used to generate the signature.
 * The assertion signature.

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

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

5.2.3. The authenticatorCancel operation

This operation takes no input parameters and returns no result.

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

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

5.3. Credential Attestation

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

An important component of the attestation object is the credential attestation statement. This is a specific type of signed data object, containing statements about a credential itself and the authenticator that created it. It contains an attestation signature created using the key of the attesting authority (except for the case of self attestation, when it is created using the private key associated with the credential). In order to correctly interpret an attestation statement, a Relying Party needs to understand two aspects of the attestation:
 1. The attestation statement format is the manner in which the signature is represented and the various contextual bindings are incorporated into the attestation statement by the authenticator. In other words, this defines the syntax of the statement. Various existing devices and platforms (such as 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 5.3.2 Attestation Statement Formats.
 2. The attestation type defines the semantics of the attestation statement and its underlying trust model. It defines how a Relying

[fido-signature-formats-figure2.html] Generating a signature on the authenticator.

On successful completion, the authenticator returns to the user agent:
 * The identifier of the credential used to generate the signature.
 * The authenticator data used to generate the signature.
 * The assertion signature.

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

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

5.2.3. The authenticatorCancel operation

This operation takes no input parameters and returns no result.

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

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

5.3. Credential Attestation

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

An important component of the attestation object is the credential attestation statement. This is a specific type of signed data object, containing statements about a credential itself and the authenticator that created it. It contains an attestation signature created using the key of the attesting authority (except for the case of self attestation, when it is created using the private key associated with the credential). In order to correctly interpret an attestation statement, a Relying Party needs to understand two aspects of the attestation:
 1. The attestation statement format is the manner in which the signature is represented and the various contextual bindings are incorporated into the attestation statement by the authenticator. In other words, this defines the syntax of the statement. Various existing devices and platforms (such as 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 5.3.2 Attestation Statement Formats.
 2. The attestation type defines the semantics of the attestation statement and its underlying trust model. It defines how a Relying

Party establishes trust in a particular attestation statement, after verifying that it is cryptographically valid. This specification supports a number of attestation types, as described in 5.3.3 Attestation Types.

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

The privacy, security and operational characteristics of attestation 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.

5.3.1. Attestation data

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

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

```
pubKey = $pubKeyFmt

; All public key formats must include an alg name
pubKeyTemplate = { alg: text }
pubKeyTemplate .within $pubKeyFmt

pubKeyFmt /= rsaPubKey
rsaPubKey = { alg: rsaAlgName, n: biguint, e: uint }
rsaAlgName = "RS256" / "RS384" / "RS512" / "PS256" / "PS384" / "PS51
2"

pubKeyFmt /= eccPubKey
eccPubKey = { alg: eccAlgName, x: biguint, y: biguint }
eccAlgName = "ES256" / "ES384" / "ES512"
```

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

5.3.2. Attestation Statement Formats

As described above, an attestation statement format is a data format which represents a cryptographic signature by an authenticator over a

Party establishes trust in a particular attestation statement, after verifying that it is cryptographically valid. This specification supports a number of attestation types, as described in 5.3.3 Attestation Types.

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

The privacy, security and operational characteristics of attestation 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 Party will also need to understand the characteristics of the authenticators that they trust, based on information they have about these authenticators. For example, the FIDO Metadata Service [FIDOMetadataService] provides one way to access such information.

5.3.1. Attestation data

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

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

```
pubKey = $pubKeyFmt

; All public key formats must include an alg name
pubKeyTemplate = { alg: text }
pubKeyTemplate .within $pubKeyFmt

pubKeyFmt /= rsaPubKey
rsaPubKey = { alg: rsaAlgName, n: biguint, e: uint }
rsaAlgName = "RS256" / "RS384" / "RS512" / "PS256" / "PS384" / "PS51
2"

pubKeyFmt /= eccPubKey
eccPubKey = { alg: eccAlgName, x: biguint, y: biguint }
eccAlgName = "ES256" / "ES384" / "ES512"
```

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

5.3.2. Attestation Statement Formats

As described above, an attestation statement format is a data format which represents a cryptographic signature by an authenticator over a

set of contextual bindings. Each attestation statement format is
defined by the following attributes:
 * Its attestation statement format identifier.
 * The set of attestation types supported by the format.
 * The syntax of an attestation statement produced in this format,
   defined using CDDL for the extension point $attStmtFormat defined
   in 5.3.4 Generating an Attestation Object.
 * The procedure for computing an attestation statement in this format
   given the credential to be attested, the authenticatorData for the
   attestation, and a clientDataHash.

 * The procedure for verifying an attestation statement, which takes
   as inputs the authenticatorData claimed to have been used for the
   attestation and the clientDataHash of the client's contextual
   bindings, and returns either:
    + An error indicating that the attestation is invalid, or
    + The attestation type, and the trust path of the attestation.
      This trust path is either empty (in case of self-attestation),
      a DAA root key (in the case of Direct Anonymous Attestation),
      or a set of X.509 certificates.

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

5.3.3. Attestation Types

WebAuthn supports multiple attestation types:

Basic Attestation

In the case of basic attestation [UAFProtocol], the authenticator's attestation key pair is specific to an authenticator model. Thus, authenticators of the same model often share the same attestation key pair. See 5.3.5.1 Privacy for futher information.

Self Attestation

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

Privacy CA

In this case, the Authenticator owns an authenticator-specific (endorsement) key. This key is used to securely communicate with a trusted third party, the Privacy CA. The Authenticator can generate multiple attestation key pairs and asks the Privacy CA to issue an attestation certificate for it. Using this approach, the Authenticator can limit the exposure of the endorsement key (which is a global correlation handle) to Privacy CA(s). Attestation keys can be requested for each scoped credential individually.

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

Direct Anonymous Attestation (DAA)

In this case, the Authenticator receives DAA credentials from a single DAA-Issuer. These DAA credentials are used along with blinding to sign the attestation data. The concept of blinding avoids the DAA credentials being misused as global correlation handle. WebAuthn supports DAA using elliptic curve cryptography and bilinear pairings, called ECDAA (see [FIDOEcdaaAlgorithm]) in this specification.

set of contextual bindings. Each attestation statement format is
defined by the following attributes:
 * Its attestation statement format identifier.
 * The set of attestation types supported by the format.
 * The syntax of an attestation statement produced in this format,
   defined using CDDL for the extension point $attStmtFormat defined
   in 5.3.4 Generating an Attestation Object.
 * The procedure for computing an attestation statement in this format
   given the credential to be attested, the authenticator data
   structure containing the authenticator data for the attestation,
   and the hash of the serialized client data.
 * The procedure for verifying an attestation statement, which takes
   as inputs the authenticator data structure containing the
   authenticator data claimed to have been used for the attestation
   and the hash of the serialized client data, and returns either:
    + An error indicating that the attestation is invalid, or
    + The attestation type, and the trust path of the attestation.
      This trust path is either empty (in case of self-attestation),
      an identifier of a ECDAA-Issuer public key (in the case of
      ECDAA), or a set of X.509 certificates.

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

5.3.3. Attestation Types

WebAuthn supports multiple attestation types:

Basic Attestation

In the case of basic attestation [UAFProtocol], the authenticator's attestation key pair is specific to an authenticator model. Thus, authenticators of the same model often share the same attestation key pair. See 5.3.5.1 Privacy for futher information.

Self Attestation

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

Privacy CA

In this case, the Authenticator owns an authenticator-specific (endorsement) key. This key is used to securely communicate with a trusted third party, the Privacy CA. The Authenticator can generate multiple attestation key pairs and asks the Privacy CA to issue an attestation certificate for it. Using this approach, the Authenticator can limit the exposure of the endorsement key (which is a global correlation handle) to Privacy 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 attestation data. The concept of blinding avoids the DAA credentials being misused as global correlation handle. WebAuthn supports DAA using elliptic curve cryptography and bilinear pairings, called ECDAA (see [FIDOEcdaaAlgorithm]) in this specification. Consequently we denote the DAA-Issuer as

5.3.4. Generating an Attestation Object

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

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

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

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

```
attObj = {
            authData: bytes,
            $$attStmtType
         }

attStmtTemplate = (
                      fmt: text,
                      attStmt: bytes
                  )

; Every attestation statement format must have the above fields
attStmtTemplate .within $$attStmtType
```

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

fmt

The attestation statement format identifier associated with the attestation statement. Each attestation statement format defines its identifier.

authData

The authenticator data used to generate the attestation statement.

attStmt

The attestation statement constructed above. The syntax of this is defined by the attestation statement format used.

5.3.5. Security Considerations

5.3.5.1. Privacy

Attestation keys may be used to track users or link various online identities of the same user together. This may be mitigated in several ways, including:
* A WebAuthn Authenticator manufacturer may choose to ship all of their devices with the same (or a fixed number of) attestation key(s) (called Basic Attestation). This will anonymize the user at the risk of not being able to revoke a particular attestation key should its WebAuthn Authenticator be compromised.
* A WebAuthn Authenticator may be capable of dynamically generating different attestation keys (and requesting related certificates) per origin (following the Privacy CA approach). For example, a WebAuthn Authenticator can ship with a master attestation key (and certificate), and combined with a cloud operated privacy CA, can dynamically generate per origin attestation keys and attestation certificates.
* A WebAuthn Authenticator can implement direct anonymous attestation (see [FIDOEcdaaAlgorithm]). Using this scheme, the authenticator

ECDAA-Issuer (see [FIDOEcdaaAlgorithm]).

5.3.4. Generating an Attestation Object

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

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

The authenticator MUST then run the signing procedure for the desired attestation statement format with this authenticator data and the hash of the serialized client data as input, and use this to construct an attestation statement in that attestation statement format.

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

```
attObj = {
            authData: bytes,
            $$attStmtType
         }

attStmtTemplate = (
                      fmt: text,
                      attStmt: bytes
                  )

; Every attestation statement format must have the above fields
attStmtTemplate .within $$attStmtType
```

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

fmt

The attestation statement format identifier associated with the attestation statement. Each attestation statement format defines its identifier.

authData

The authenticator data used to generate the attestation statement.

attStmt

The attestation statement constructed above. The syntax of this is defined by the attestation statement format used.

5.3.5. Security Considerations

5.3.5.1. Privacy

Attestation keys may be used to track users or link various online identities of the same user together. This may be mitigated in several ways, including:
* A WebAuthn authenticator manufacturer may choose to ship all of their devices with the same (or a fixed number of) attestation key(s) (called Basic Attestation). This will anonymize the user at the risk of not being able to revoke a particular attestation key should its WebAuthn Authenticator be compromised.
* A WebAuthn Authenticator may be capable of dynamically generating different attestation keys (and requesting related certificates) per origin (following the Privacy CA approach). For example, a WebAuthn Authenticator can ship with a master attestation key (and certificate), and combined with a cloud operated privacy CA, can dynamically generate per origin attestation keys and attestation certificates.
* A WebAuthn Authenticator can implement Elliptic Curve based direct anonymous attestation (see [FIDOEcdaaAlgorithm]). Using this

generates a blinded attestation signature. This allows the Relying Party to verify the signature using the DAA root key, but the attestation signature doesn't serve as a global correlation handle.

5.3.5.2. Attestation Certificate and Attestation Certificate CA Compromise

When an intermediate CA or a root CA used for issuing attestation certificates is compromised, WebAuthn Authenticator attestation keys are still safe although their certificates can no longer be trusted. A WebAuthn Authenticator manufacturer 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") scoped credentials that were registered after the CA compromise date using an attestation certificate chaining up to the same intermediate CA. It is thus recommended that Relying Parties remember intermediate attestation CA certificates during Authenticator registration in order to un-register related Scoped Credentials if the registration was performed after revocation of such certificates.

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

5.3.5.3. Attestation Certificate Hierarchy

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 authenticatorData.

6. Relying Party Operations

Upon successful execution of a makeCredential() or getAssertion() call, the Relying Party's script receives a ScopedCredentialInfo or AuthenticationAssertion structure respectively from the client. It must then deliver the contents of this structure to the Relying Party, using methods outside the scope of this specification. This section describes the operations that the Relying Party must perform upon receipt of these structures.

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 doesn't serve as a global correlation handle.

5.3.5.2. Attestation Certificate and Attestation Certificate CA Compromise

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

5.3.5.3. Attestation Certificate Hierarchy

A 3-tier hierarchy for attestation certificates is recommended (i.e., Attestation Root, Attestation Issuing CA, Attestation Certificate). It is also recommended that for each WebAuthn Authenticator device line (i.e., model), a separate issuing CA is used to help 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.

6. 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.

6.1. Registering a new credential

When requested to register a new credential, represented by a ScopedCredentialInfo structure, as part of a registration ceremony, a Relying Party MUST proceed as follows:

1. Perform JSON deserialization on the clientDataJSON field of the ScopedCredentialInfo object to extract the ClientData C claimed to have been used for the credential's attestation.
2. Verify that the challenge in C matches the challenge that was sent to the authenticator in the makeCredential() call.
3. Verify that the origin in C matches the Relying Party's origin.
4. Verify that the tokenBinding in C matches the token binding ID for the TLS connection over which the attestation was obtained.
5. Verify that the extensions in C is a proper subset of the extensions requested by the RP.
6. Compute the clientDataHash over clientDataJSON using the hashAlg algorithm found in C.
7. Perform CBOR decoding on the attestationObject field of the ScopedCredentialInfo structure to obtain the attestation statement format fmt, the authenticator data authData, and the attestation statement attStmt.
8. Verify that the RP ID hash in authData is indeed the SHA-256 hash of the RP ID expected by the RP.
9. Determine the attestation statement format by performing an ASCII case-insensitive match on fmt against the set of WebAuthn Attestation Statement Format Identifiers given in the IANA Registry of the same name [WebAuthn-Registries].
10. Verify that attStmt is a correct, validly-signed attestation statement, using the attestation statement format fmt's verification procedure given authenticator data authData and the clientDataHash computed in step 6.
11. If validation is successful, obtain a list of acceptable trust anchors (attestation root certificates or DAA root keys) for that attestation type and attestation statement format fmt, from a trusted source or from policy. For example, the FIDO Metadata Service [FIDOMetadataService] provides one way to obtain such information, using the AAGUID in the attestation data contained in authData.
12. Assess the attestation trustworthiness using the outputs of the verification procedure in step 10, as follows:
    + If self-attestation was used, check if self-attestation is acceptable under Relying Party policy.
    + If DAA was used, verify that the DAA key used is in the set of acceptable trust anchors obtained in step 11.
    + Otherwise, use the X.509 certificates returned by the verification procedure to verify that the attestation public key correctly chains up to an acceptable root certificate.
13. If the attestation statement attStmt verified successfully and is found to be trustworthy, then register the new credential with the account that was denoted in the accountInformation passed to makeCredential(), by associating it with the credential ID and credential public key contained in authData's attestation data, as appropriate for the Relying Party's systems.
14. If the attestation statement attStmt successfully verified but is not trustworthy per step 12 above, the Relying Party SHOULD fail the registration ceremony.
    NOTE: However, if permitted by policy, the Relying Party MAY register the credential ID and credential public key but treat the credential as one with self-attestation (see 5.3.3 Attestation Types). If doing so, the Relying Party is asserting there is no cryptographic proof that the Scoped Credential has been generated by a particular Authenticator model. See [FIDOSecRef] and [UAFProtocol] for a more detailed discussion.
15. If verification of the attestation statement failed, the Relying Party MUST fail the registration ceremony.

6.1. 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 challenge in C matches the challenge that was sent to the authenticator in the create() call.
3. Verify that the origin in C matches the Relying Party's origin.
4. Verify that the tokenBinding in C matches the Token Binding ID for the TLS connection over which the attestation was obtained.
5. Verify that the clientExtensions in C is a proper subset of the extensions requested by the RP and that the authenticatorExtensions in C is also a proper subset of the extensions requested by the RP.
6. Compute the hash of clientDataJSON using the algorithm identified by C.hashAlg.
7. 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.
8. Verify that the RP ID hash in authData is indeed the SHA-256 hash of the RP ID expected by the RP.
9. Determine the attestation statement format by performing an 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].
10. Verify that attStmt is a correct, validly-signed attestation statement, using the attestation statement format fmt's verification procedure given authenticator data authData and the hash of the serialized client data computed in step 6.
11. 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 attestation data contained in authData.
12. 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 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.
13. If the attestation statement attStmt verified successfully and is found to be trustworthy, then register the new credential with the account that was denoted in the options.user passed to create(), by associating it with the credential ID and credential public key contained in authData's attestation data, as appropriate for the Relying Party's systems.
14. If the attestation statement attStmt successfully verified but is not trustworthy per step 12 above, the Relying Party SHOULD fail the registration ceremony.
    NOTE: However, if permitted by policy, the Relying Party MAY register the credential ID and credential public key but treat the credential as one with self-attestation (see 5.3.3 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.
15. If verification of the attestation statement failed, the Relying Party MUST fail the registration ceremony.

Verification of attestation objects requires that the Relying Party has a trusted method of determining acceptable trust anchors in step 11 above. Also, if certificates are being used, the Relying Party must have access to certificate status information for the intermediate CA certificates. The Relying Party must also be able to build the attestation certificate chain if the client did not provide this chain in the attestation information.

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

6.2. Verifying an authentication assertion

When requested to authenticate a given AuthenticationAssertion structure as part of an authentication ceremony, the Relying Party MUST proceed as follows:

1. Using the id attribute contained in the credential attribute of the given AuthenticationAssertion structure, look up the corresponding credential public key.
2. Let cData, aData and sig denote the clientDataJSON, authenticatorData and signature attributes of the given AuthenticationAssertion structure, respectively.
3. Perform JSON deserialization on cData to extract the ClientData C used for the signature.
4. Verify that the challenge member of C matches the challenge that was sent to the authenticator in the getAssertion() call.
5. Verify that the origin member of C matches the Relying Party's origin.
6. Verify that the tokenBinding member of C (if present) matches the token binding ID for the TLS connection over which the signature was obtained.
7. Verify that the extensions member of C is a proper subset of the extensions requested by the RP.
8. Verify that the RP ID hash in aData is the SHA-256 hash of the RP ID expected by the RP.
9. Let hash be the result of computing a hash over the cData using the algorithm represented by the hashAlg member of C.
10. 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.
11. If all the above steps are successful, continue with the authentication ceremony as appropriate. Otherwise, fail the authentication ceremony.

7. Defined Attestation Statement Formats

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

7.1. Attestation Statement Format Identifiers

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

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 reverse domain-name naming, using a domain name registered by the

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.

6.2. 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 challenge member of C matches the challenge that was sent to the authenticator in the PublicKeyCredentialRequestOptions passed to the get() call.
5. Verify that the origin member of C matches the Relying Party's origin.
6. Verify that the tokenBinding member of C (if present) matches the Token Binding ID for the TLS connection over which the signature was obtained.
7. Verify that the clientExtensions member of C is a proper subset of the extensions requested by the Relying Party and that the authenticatorExtensions in C is also a proper subset of the extensions requested by the Relying Party.
8. Verify that the RP ID hash in aData is the SHA-256 hash of the RP ID expected by the Relying Party.
9. Let hash be the result of computing a hash over the cData using the algorithm represented by the hashAlg member of C.
10. 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.
11. If all the above steps are successful, continue with the authentication ceremony as appropriate. Otherwise, fail the authentication ceremony.

7. Defined Attestation Statement Formats

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

7.1. Attestation Statement Format Identifiers

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

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 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, i.e., VCHAR as defined in [RFC5234] (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-insensitive 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. See the WebAuthn Attestation Statement Format Identifier Registry defined in [WebAuthn-Registries] for an up-to-date list of registered attestation statement formats.

7.2. Packed Attestation Statement Format

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

Attestation statement format identifier

packed

Attestation types supported

All

Syntax

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

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

packedStmtFormat = {
                       alg: rsaAlgName / eccAlgName,
                       sig: bytes,
                       x5c: [ attestnCert: bytes, * (caCert: bytes) ]
                   } //
                   {
                       alg: "ED256" / "ED512",
                       sig: bytes,
                       daaKey: eccPubKey
                   }
```

The semantics of the fields are as follows:

alg

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

sig

A byte string containing the attestation signature.

x5c

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].

7.2. Packed Attestation Statement Format

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

Attestation statement format identifier

packed

Attestation types supported

All

Syntax

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

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

packedStmtFormat = {
                       alg: rsaAlgName / eccAlgName,
                       sig: bytes,
                       x5c: [ attestnCert: bytes, * (caCert: bytes) ]
                   } //
                   {
                       alg: "ED256" / "ED512",
                       sig: bytes,
                       ecdaaKeyId: bytes
                   }
```

The semantics of the fields are as follows:

alg

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

sig

A byte string containing the attestation signature.

x5c

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

daaKey

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

Signing procedure

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

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

If DAA is in use, the authenticator produces sig by concatenating the given authenticatorData and clientDataHash, and signing the result using DAA-Sign with a DAA root key selected through an authenticator-specific mechanism (see [FIDOEcdaaAlgorithm]). It sets alg to the algorithm of the DAA root key and daaKey to the DAA root key.

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

Verification procedure

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

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

+ Verify that sig is a valid signature over the concatenation of the given authenticatorData and clientDataHash using the attestation public key in x5c with the algorithm specified in alg.
+ Verify that x5c meets the requirements in 7.2.1 Packed attestation statement certificate requirements.
+ If x5c contains an extension with OID 1 3 6 1 4 1 45724 1 1 4 (id-fido-gen-ce-aaguid) verify that the value of this extension matches the AAGUID in the claimed authenticatorData.
+ If successful, return attestation type Basic and trust path x5c.

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

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

ecdaaKeyId

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

Signing procedure

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

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

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

If ECDAA is in use, the authenticator produces sig by concatenating authenticatorData and clientDataHash, and signing the result using ECDAA-Sign (see section 3.5 of [FIDOEcdaaAlgorithm]) with a ECDAA-Issuer public key selected through an authenticator-specific mechanism (see [FIDOEcdaaAlgorithm]). It sets alg to the algorithm of the ECDAA-Issuer public key and ecdaaKeyId to the identifier of the ECDAA-Issuer public key (see above).

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

Verification procedure

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

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

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

+ Verify that sig is a valid signature over the concatenation of authenticatorData and clientDataHash using the attestation public key in x5c with the algorithm specified in alg.
+ Verify that x5c meets the requirements in 7.2.1 Packed attestation statement certificate requirements.
+ If x5c contains an extension with OID 1 3 6 1 4 1 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 Basic and trust path x5c.

If ecdaaKeyId is present, then the attestation type is ECDAA. In this case:

+ Verify that sig is a valid signature over the concatenation of
  the given authenticatorData and clientDataHash using
  DAA-Verify with daaKey (see [FIDOEcdaaAlgorithm]).
+ If successful, return attestation type DAA and trust path
  daaKey.

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

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

7.2.1. Packed attestation statement certificate requirements

The attestation certificate MUST have the following fields/extensions:
* Version must be set to 3.
* Subject field MUST be set to:

Subject-C

Country where the Authenticator vendor is incorporated

Subject-O

Legal name of the Authenticator vendor

Subject-OU

Authenticator Attestation

Subject-CN

No stipulation.

* If the related attestation root certificate is used for multiple
  authenticator models, the Extension OID 1 3 6 1 4 1 45724 1 1 4
  (id-fido-gen-ce-aaguid) MUST be present, containing the AAGUID as
  value.
* The Basic Constraints extension MUST have the CA component set to
  false
* An Authority Information Access (AIA) extension with entry
  id-ad-ocsp and a CRL Distribution Point extension [RFC5280] are
  both optional as the status of many attestation certificates is
  available through authenticator metadata services. See, for
  example, the FIDO Metadata Service [FIDOMetadataService].

7.3. TPM Attestation Statement Format

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

Attestation statement format identifier

tpm

Attestation types supported

Privacy CA, DAA

Syntax

The syntax of a TPM Attestation statement is as follows:

+ Verify that sig is a valid signature over the concatenation of
  authenticatorData and clientDataHash using ECDAA-Verify with
  ECDAA-Issuer public key identified by ecdaaKeyId (see
  [FIDOEcdaaAlgorithm]).
+ If successful, return attestation type ECDAA and trust path
  ecdaaKeyId.

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

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

7.2.1. Packed attestation statement certificate requirements

The attestation certificate MUST have the following fields/extensions:
* Version must be set to 3.
* Subject field MUST be set to:

Subject-C

Country where the Authenticator vendor is incorporated

Subject-O

Legal name of the Authenticator vendor

Subject-OU

Authenticator Attestation

Subject-CN

No stipulation.

* If the related attestation root certificate is used for multiple
  authenticator models, the Extension OID 1 3 6 1 4 1 45724 1 1 4
  (id-fido-gen-ce-aaguid) MUST be present, containing the AAGUID as
  value.
* The Basic Constraints extension MUST have the CA component set to
  false
* An Authority Information Access (AIA) extension with entry
  id-ad-ocsp and a CRL Distribution Point extension [RFC5280] are
  both optional as the status of many attestation certificates is
  available through authenticator metadata services. See, for
  example, the FIDO Metadata Service [FIDOMetadataService].

7.3. TPM Attestation Statement Format

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

Attestation statement format identifier

tpm

Attestation types supported
Privacy CA, ECDAA

Syntax

The syntax of a TPM Attestation statement is as follows:

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

tpmStmtFormat = {
                    ver: "2.0",
                    (
                        alg: rsaAlgName / eccAlgName,
                        x5c: [ aikCert: bytes, * (caCert: bytes) ]
                    ) //
                    (
                        alg:  "ED256" / "ED512",
                        daaKey: eccPubKey
                    ),
                    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

The name of the algorithm used to generate the attestation signature. The types rsaAlgName and eccAlgNAme are as defined in 5.3.1 Attestation data. The types "ED256" and "ED512" refer to the algorithms specified in [FIDOEcdaaAlgorithm].

x5c

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

daaKey

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

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

Concatenate the given authenticatorData and clientDataHash to form attToBeSigned.


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

tpmStmtFormat = {
                    ver: "2.0",
                    (
                        alg: rsaAlgName / eccAlgName,
                        x5c: [ aikCert: bytes, * (caCert: bytes) ]
                    ) //
                    (
                        alg:  "ED256" / "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

The name of the algorithm used to generate the attestation signature. The types rsaAlgName and eccAlgNAme are as defined in 5.3.1 Attestation data. The types "ED256" and "ED512" refer to the algorithms specified in [FIDOEcdaaAlgorithm].

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 qualifyingData parameter to attToBeSigned.

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

Verification procedure

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

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

Concatenate the given authenticatorData and clientDataHash to form attToBeSigned.

Validate that certInfo is valid:

+ Verify that magic is set to TPM_GENERATED_VALUE.
+ Verify that type is set to TPM_ST_ATTEST_CERTIFY.
+ Verify that extraData is set to attToBeSigned.
+ Verify that attested contains a TPMS_CERTIFY_INFO structure, whose name field contains a valid Name for pubArea, as computed using the algorithm in the 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 DAA. In this case:

+ Verify the sig is a valid signature over certInfo using the attestation public key in x5c with the algorithm specified in alg.
+ Verify that x5c meets the requirements in 7.3.1 TPM attestation statement certificate requirements.
+ If x5c contains an extension with OID 1 3 6 1 4 1 45724 1 1 4 (id-fido-gen-ce-aaguid) verify that the value of this extension matches the AAGUID in the claimed authenticatorData.
+ If successful, return attestation type Privacy CA and trust path x5c.

If daaKey is present, then the attestation type is DAA.

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

7.3.1. TPM attestation statement certificate requirements

TPM attestation certificate MUST have the following fields/extensions:
* Version must be set to 3.
* Subject field MUST be set to empty.
* The Subject Alternative Name extension must be set as defined in [TPMv2-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

Generate a signature using the procedure specified in [TPMv2-Part3] Section 18.2, using the attestation private key and setting the qualifyingData parameter to attToBeSigned.

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

Verification procedure

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

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

Verify that the public key specified by the parameters and unique fields of pubArea is identical to the public key contained in the attestation data inside 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 attToBeSigned.
+ Verify that attested contains a TPMS_CERTIFY_INFO structure, whose name field contains a valid Name for pubArea, as computed using the algorithm in the 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 7.3.1 TPM attestation statement certificate requirements.
+ If x5c contains an extension with OID 1 3 6 1 4 1 45724 1 1 4 (id-fido-gen-ce-aaguid) verify that the value of this extension matches the AAGUID in authenticatorData.
+ If successful, return attestation type Privacy CA and 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.

7.3.1. TPM attestation statement certificate requirements

TPM attestation certificate MUST have the following fields/extensions:
* Version must be set to 3.
* Subject field MUST be set to empty.
* The Subject Alternative Name extension must be set as defined in [TPMv2-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].

7.4. Android Key Attestation Statement Format

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

Attestation statement format identifier

android-key

Attestation types supported

Basic

Syntax

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

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

androidStmtFormat = bytes
```

Signing procedure

Concatenate the given authenticatorData and clientDataHash to form attToBeSigned.

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

Verification procedure

Verification is performed as follows:

+ Verify that the public key in the first certificate in the series of certificates represented by the signature matches the credential public key in the attestation data field of the given authenticatorData.
+ Verify that in the attestation certificate extension data:
  o The value of the attestationChallenge field is identical to the concatenation of the claimed authenticatorData and clientDataHash.
  o The AuthorizationList.allApplications field is not present, since ScopedCredentials must be bound to the RP

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].

7.4. Android Key Attestation Statement Format

When the authenticator in question is a platform-provided Authenticator on the Android "N" or later platform, the attestation statement is based on the Android key attestation. In these cases, the attestation statement is produced by a component running in a secure operating environment, but the 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 = bytes
```

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

Verification procedure

Verification is performed as follows:

+ Let authenticatorData denote the authenticator data claimed to have been used for the attestation, and let clientDataHash denote the hash of the serialized client data.
+ Verify that the public key in the first certificate in the series of certificates represented by the signature matches the credential public key in the attestation data field of 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

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 trust path set to the entire attestation statement.

7.5. Android SafetyNet Attestation Statement Format

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

Attestation statement format identifier

android-safetynet

Attestation types supported

Basic

Syntax

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

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

Signing procedure

Concatenate the given authenticatorData and clientDataHash to form attToBeSigned.

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

Verification procedure

Verification is performed as follows:

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 trust path set to the entire attestation statement.

7.5. Android SafetyNet Attestation Statement Format

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

Attestation statement format identifier

android-safetynet

Attestation types supported

Basic

Syntax

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

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

Signing procedure

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

Concatenate authenticatorData and clientDataHash to form attToBeSigned.

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

Verification procedure

Verification is performed as follows:

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

7.6. FIDO U2F Attestation Statement Format

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

Attestation statement format identifier

fido-u2f

Attestation types supported

Basic

Syntax

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

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

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

The semantics of the above fields are as follows:

x5c

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

sig

The attestation signature.

Signing procedure

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

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

Generate a signature as specified in [FIDO-U2F-Message-Formats] section 4.3, with the application parameter set to the SHA-256

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

7.6. FIDO U2F Attestation Statement Format

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

Attestation statement format identifier

fido-u2f

Attestation types supported

Basic

Syntax

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

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

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

The semantics of the above fields are as follows:

x5c

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

sig

The attestation signature.

Signing procedure

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

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

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

Generate a signature as specified in [FIDO-U2F-Message-Formats] section 4.3, with the application parameter set to the SHA-256 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 this as sig and set the attestation certificate of the attestation public key as x5c.

Verification procedure

Verification is performed as follows:

+ Verify that the given attestation statement is valid CBOR conforming to the syntax defined above.
+ If x5c is not a certificate for an ECDSA public key over the P-256 curve, stop verification and return an error.
+ If the given clientDataHash is 256 bits long, set tbsHash to this value. Otherwise set tbsHash to the SHA-256 hash of the given clientDataHash.
+ From the given authenticatorData, extract the claimed RP ID hash, the claimed credential ID and the claimed credential public key.

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

8. WebAuthn Extensions

The mechanism for generating scoped credentials, as well as requesting and generating Authentication assertions, as defined in 4 Web Authentication API, can be extended to suit particular use cases. Each case is addressed by defining a registration extension and/or an authentication extension. Extensions can define additions to the following steps and data:
* makeCredential() request parameters for registration extension.
* getAssertion() request parameters for authentication extensions.
* Client processing, and the ClientData structure, for registration extensions and authentication extensions.
* Authenticator processing, and the authenticatorData structure, for registration extensions and authentication extensions.

When requesting an assertion for a scoped credential, a Relying Party can list a set of extensions to be used, if they are supported by the client and/or the authenticator. It sends the client arguments for each extension in the getAssertion() call (for authentication extensions) or makeCredential() call (for registration extensions) to the client platform. The client platform performs additional processing for each extension that it supports, and augments ClientData as required by the extension. In addition, the client collects the authenticator arguments for the above extensions, and passes them to the authenticator in the authenticatorMakeCredential call (for registration extensions) or authenticatorGetAssertion call (for authentication extensions). These authenticator arguments are passed as name-value pairs, with the

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 this as sig and set the attestation certificate of the attestation public key as x5c.

Verification procedure

Verification is performed as follows:

+ Verify that the given attestation statement is valid CBOR conforming to the syntax defined above.
+ If x5c is not a certificate for an ECDSA public key over the P-256 curve, stop verification and return an error.
+ Let authenticatorData denote the authenticator data claimed to have been used for the attestation, and let clientDataHash denote the hash of the serialized client data.
+ If clientDataHash is 256 bits long, set tbsHash to this value. Otherwise set tbsHash to the SHA-256 hash of clientDataHash.
+ From authenticatorData, extract the claimed RP ID hash, the claimed credential ID and the claimed credential public key.
+ Generate the claimed to-be-signed data as specified in [FIDO-U2F-Message-Formats] section 4.3, with the application parameter set to the claimed RP ID hash, the challenge parameter set to tbsHash, the key handle parameter set to the claimed credential ID of the given credential, and the user public key parameter set to the claimed credential public key.
+ Verify that the sig is a valid ECDSA P-256 signature over the to-be-signed data constructed above.
+ If successful, return attestation type Basic with the trust path set to x5c.

8. WebAuthn Extensions

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

Every extension is a client extension, meaning that the extension involves communication with and processing by the client. Client extensions define the following steps and data:
* navigator.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 extension identifier as the name, and the corresponding authenticator argument as the value. The authenticator, in turn, performs additional processing for the extensions that it supports, and augments authenticatorData as specified by the extension.

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

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

8.1. Extension Identifiers

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

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

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

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

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

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

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

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

8.1. Extension Identifiers

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

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

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

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

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

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

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

8.2. Defining extensions

A definition of an extension must specify, at minimum, an extension identifier and an extension client argument sent via the getAssertion() or makeCredential() call. Additionally, extensions may specify additional values in ClientData, authenticatorData (in the case of authentication extensions), or both. Finally, if the extension requires any authenticator processing, it must also specify an authenticator argument to be sent via the authenticatorGetAssertion or authenticatorMakeCredential call.

Any extension that requires client processing MUST specify a method of augmenting ClientData that unambiguously lets the Relying Party know that the extension was honored by the client. Similarly, any extension that requires authenticator processing MUST specify a method of augmenting authenticatorData to let the Relying Party know that the extension was honored by the authenticator.

8.3. Extending request parameters

An extension defines up to two request arguments. The client argument is passed from the Relying Party to the client in the getAssertion() or makeCredential() call, while the authenticator argument is passed from the client to the authenticator during the processing of these calls.

A Relying Party simultaneously requests the use of an extension and sets its client argument by including an entry in the extensions option to the makeCredential() or getAssertion() call. The entry key MUST be the extension identifier, and the value MUST be the client argument.

```
var assertionPromise = credentials.getAssertion(..., /* extensions */ {
  "webauthnExample_foobar": 42
});
```

Extension definitions MUST specify the valid values for their client argument. Clients SHOULD ignore extensions with an invalid client argument. 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.

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

9 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.

8.2. 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.

8.3. 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: {
    challenge: "...",
    extensions: {
      "webauthnExample_foobar": 42
    }
  }
});
```

Extension definitions MUST specify the valid values for their client extension input. Clients SHOULD ignore extensions with an invalid client extension input. If an extension does not require any parameters from the Relying Party, it SHOULD be defined as taking a Boolean client argument, set to true to signify that the extension is requested by the Relying Party.

Extensions that only affect client processing need not specify an authenticator argument. Extensions that affect authenticator processing MUST specify a method of computing the authenticator argument from the client argument. For extensions that do not require additional parameters, and are defined as taking a Boolean client argument set to true, this method SHOULD consist of passing an authenticator argument of true (CBOR major type 7, value 21).

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

8.4. Extending client processing

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

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

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

8.5. Extending authenticator processing

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

As specified in 5.1 Authenticator data, the authenticator data value of each processed extension is included in the extended data part of the authenticatorData. This part is a CBOR map, with extension identifiers as keys, and the authenticator data value of each extension as the value.

8.6. Example extension

This section is not normative.

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

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

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.

8.4. 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 clientExtensionResults 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.

8.5. Authenticator extension processing

As specified in 5.1 Authenticator data, the CBOR authenticator extension input value of each processed authenticator extension is included in the extensions data part of the authenticator data. 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.

8.6. 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 =
   credentials.getAssertion("SGFulFNvbG8gc2hvdCBmaXJzdC4",
       {}, /* Empty filter */
       { 'webauthnExample_geo': true });
```

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

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

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

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

```
81 (hex)                                    -- Flags, ED and TUP both set.
20 05 58 1F                                   -- Signature counter
A1                                        -- CBOR map of one element
    73                                    -- Key 1: CBOR text string of 19 byt
es
        77 65 62 61 75 74 68 6E 45 78 61
        6D 70 6C 65 5F 67 65 6F             -- "webauthnExample_geo" UTF-8 encod
ed 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
```

9. Pre-defined extensions

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

9.1. FIDO AppId

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


```
var assertionPromise =
   navigator.credentials.get({
       publicKey: {
           challenge: "SGFulFNvbG8gc2hvdCBmaXJzdC4",
           allowList: [], /* 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 TUP both set.
20 05 58 1F                                   -- Signature counter
A1                                        -- CBOR map of one element
    73                                    -- Key 1: CBOR text string of 19 byt
es
        77 65 62 61 75 74 68 6E 45 78 61
        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]
       }
   }
}
```

9. 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.

9.1. 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 getAssertion() call; other usage will result in client error.

Extension identifier

fido_appid

Client argument

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

Client processing

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

Authenticator argument

none

Authenticator processing

none

Authenticator data

none

9.2. Transaction authorization

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

Extension identifier

webauthn_txAuthSimple

Client argument

A single UTF-8 encoded string prompt.

Client processing

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

Authenticator argument

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

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, reject promise with a DOMException whose name is "NotAllowedError", and terminate this algorithm. Replace the calculation of rpId in Step 3 of 4.1.4 Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) 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.

9.2. 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 processing

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

Authenticator data

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

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

Extension identifier

webauthn_txAuthGeneric

Client argument

A CBOR map defined as follows:

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

Client processing

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

Authenticator argument

The client argument encoded as a CBOR map.

Authenticator processing

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

Authenticator data

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

9.3. Authenticator Selection Extension

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

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

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

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

9.3. 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 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.

9.4. Authenticator Selection Extension (authnSel)

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

Extension identifier

webauthn_authnSel

Client argument

A sequence of AAGUIDs:

typedef sequence<AAGUID> AuthenticatorSelectionList;

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

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

typedef BufferSource AAGUID;

Client processing

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

Authenticator argument

There is no authenticator argument.

Authenticator processing

None.

9.4. SupportedExtensions Extension

Extension identifier

webauthn_exts

Client argument

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

Client processing

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

Authenticator argument

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

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

Authenticator extension input
None.

Authenticator extension processing
None.

Authenticator extension output
None.

9.5. 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 processing

The authenticator augments the authenticator data with a list of extensions that the authenticator supports, as defined below. This extension can be added to attestation objects.

Authenticator data

The SupportedExtensions extension is a list (CBOR array) of extension identifiers (UTF-8 encoded strings).

9.5. User Verification Index (UVI) Extension

Extension identifier

webauthn_uvi

Client argument

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

Client processing

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

Authenticator argument

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

Authenticator processing

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

Authenticator data

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

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

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

Servers supporting UVI extensions MUST support a length of up to

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).

9.6. 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 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 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 authenticatorData containing one UVI extension

```
...                                   -- RP ID hash (32 bytes)
81                                    -- TUP and ED set
00 00 00 01                           -- (initial) signature counter
...                                   -- all public key alg etc.
A1                                    -- extension: CBOR map of one elemen
t
   6C                                 -- Key 1: CBOR text string of 11 byt
es
      77 65 62 61 75 74 68 6E 5F 75 76 69 -- "webauthn_uvi" UTF-8 encoded stri
ng
   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
```

9.6. Location Extension

Extension identifier

webauthn_loc

Client argument

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

Client processing

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

Authenticator argument

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

Authenticator processing

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

Authenticator data

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

32 bytes for the UVI value.

Example for authenticator data containing one UVI extension

```
...                                   -- RP ID hash (32 bytes)
81                                    -- TUP 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
```

9.7. Location Extension (loc)

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

Extension identifier
loc

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

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

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

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

Authenticator extension processing

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

Authenticator extension output

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

Geolocation API Specification.

```
...                                -- RP ID hash (32 bytes)
81                                  -- TUP and ED set
00 00 00 01                             -- (initial) signature counter
...                                -- all public key alg etc.
A1                                  -- extension: CBOR map of one elemen
t
  6C                                -- Value 1: CBOR text string of 11 b
ytes
    77 65 62 61 75 74 68 6E 5F 6C 6F 63 -- "webauthn_loc" UTF-8 encoded stri
ng
  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 string
    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 string
    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 string
    FB ...                -- Element 6:  Altitude as CBOR encoded double-p
recision float
```

9.7. User Verification Mode (UVM) Extension

Extension identifier

webauthn_uvm

Client argument

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

Client processing

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

Authenticator argument

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

Authenticator processing

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

Authenticator data

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

defined in The W3C Geolocation API Specification.

```
...                                -- RP ID hash (32 bytes)
81                                  -- TUP 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
recision float
```

9.8. 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 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
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 authenticatorData containing one UVM extension for a multi-factor authentication instance where 2 factors were used:

```
...                 -- RP ID hash (32 bytes)
81                   -- TUP and ED set
00 00 00 01            -- (initial) signature counter
...                 -- all public key alg etc.
A1                   -- extension: CBOR map of one element
    6C               -- Key 1: CBOR text string of 12 bytes
        77 65 62 61 75 74 68 6E 2E 75 76 6d -- "webauthn_uvm" UTF-8 encoded stri
ng
    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. IANA Considerations

This specification registers the algorithm names "S256", "S384", "S512", and "SM3" with the IANA JSON Web Algorithms registry as defined in section "Cryptographic Algorithms for Digital Signatures and MACs" in [RFC7518].

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

Algorithm Name "S256"


```
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                   -- TUP 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. IANA Considerations

10.1. WebAuthn Attestation Statement Format Identifier Registrations

Algorithm Description   The SHA256 hash algorithm.
Algorithm Usage Location(s)   "alg", i.e., used with JWS.
JOSE Implementation Requirements Optional+
Change Controller   FIDO Alliance
Specification Documents   [FIPS-180-4]
Algorithm Analysis Document(s) [SP800-107r1]

Algorithm Name   "S384"
Algorithm Description   The SHA384 hash algorithm.
Algorithm Usage Location(s)   "alg", i.e., used with JWS.
JOSE Implementation Requirements Optional
Change Controller   FIDO Alliance
Specification Documents   [FIPS-180-4]
Algorithm Analysis Document(s) [SP800-107r1]

Algorithm Name   "S512"
Algorithm Description   The SHA512 hash algorithm.
Algorithm Usage Location(s)   "alg", i.e., used with JWS.
JOSE Implementation Requirements Optional+
Change Controller   FIDO Alliance
Specification Documents   [FIPS-180-4]
Algorithm Analysis Document(s) [SP800-107r1]

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

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

* WebAuthn Attestation Statement Format Identifier: packed
* Description: The "packed" attestation statement format is a WebAuthn-optimized format for attestation data. It uses a very compact but still extensible encoding method. This format is implementable by authenticators with limited resources (e.g., secure elements).
* Specification Document: Section 7.2 Packed Attestation Statement Format of this specification
* WebAuthn Attestation Statement Format Identifier: tpm
* Description: The TPM attestation statement format returns an attestation statement in the same format as the packed attestation statement format, although the the rawData and signature fields are computed differently.
* Specification Document: Section 7.3 TPM Attestation Statement Format of this specification
* WebAuthn Attestation Statement Format Identifier: android-key
* Description: Platform-provided authenticators based on Android versions "N", and later, may provide this proprietary "hardware attestation" statement.
* Specification Document: Section 7.4 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 7.5 Android SafetyNet Attestation Statement Format of this specification
* WebAuthn Attestation Statement Format Identifier: fido-u2f
* Description: Used with FIDO U2F authenticators
* Specification Document: Section 7.6 FIDO U2F Attestation Statement Format of this specification

10.2. WebAuthn Extension Identifier Registrations

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

* WebAuthn Extension Identifier: appid
* Description: This authentication extension allows Relying Parties that have previously registered a credential using the legacy FIDO JavaScript APIs to request an assertion.
* Specification Document: Section 9.1 FIDO AppId Extension (appid) of this specification
* WebAuthn Extension Identifier: txAuthSimple
* Description: This registration extension and authentication extension allows for a simple form of transaction authorization. A WebAuthn Relying Party can specify a prompt string, intended for display on a trusted device on the authenticator
* Specification Document: Section 9.2 Simple Transaction Authorization Extension (txAuthSimple) of this specification
* WebAuthn Extension Identifier: txAuthGeneric
* Description: This registration extension and authentication extension allows images to be used as transaction authorization

11. Sample scenarios

This section is not normative.

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

As was the case in earlier sections, this flow focuses on a use case involving an external first-factor authenticator with its own display. One example of such an authenticator would be a smart phone. Other authenticator types are also supported by this API, subject to implementation by the platform. For instance, this flow also works without modification for the case of an authenticator that is embedded in the client platform. The flow also works for the case of an 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

prompts as well. This allows authenticators without a font rendering engine to be used and also supports a richer visual appearance than accomplished with the webauthn.txauth.simple extension.
* Specification Document: Section 9.3 Generic Transaction Authorization Extension (txAuthGeneric) of this specification
* WebAuthn Extension Identifier: authnSel
* Description: This registration extension allows a WebAuthn Relying Party to guide the selection of the authenticator that will be leveraged when creating the credential. It is intended primarily for WebAuthn Relying Parties that wish to tightly control the experience around credential creation.
* Specification Document: Section 9.4 Authenticator Selection Extension (authnSel) of this specification
* WebAuthn Extension Identifier: exts
* Description: This registration extension enables the Relying Party to determine which extensions the authenticator supports. The extension data is a list (CBOR array) of extension identifiers encoded as UTF-8 Strings. This extension is added automatically by the authenticator. This extension can be added to attestation statements.
* Specification Document: Section 9.5 Supported Extensions Extension (exts) of this specification
* WebAuthn Extension Identifier: uvi
* Description: This registration extension and authentication extension enables use of a user verification index. The user verification index is a value uniquely identifying a user verification data record. The UVI data can be used by servers to understand whether an authentication was authorized by the exact same biometric data as the initial key generation. This allows the detection and prevention of "friendly fraud".
* Specification Document: Section 9.6 User Verification Index Extension (uvi) of this specification
* WebAuthn Extension Identifier: loc
* Description: 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 9.7 Location Extension (loc) of this specification
* WebAuthn Extension Identifier: uvm
* Description: This registration extension and authentication extension enables use of a user verification method. The user verification method extension returns to the Webauthn relying party which user verification methods (factors) were used for the WebAuthn operation.
* Specification Document: Section 9.8 User Verification Method Extension (uvm) of this specification

11. Sample scenarios

This section is not normative.

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

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

11.1. Registration

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

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

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

```
var webauthnAPI = navigator.authentication;

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


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

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

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


client can have information to show appropriate prompts.

11.1. Registration

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

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

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

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

var publicKey = {
  challenge: Uint8Array.from(window.atob("PGifxAoBwCkWkm4b1CiIl5otCphiIh6MijdjbW
FjomA="), c=>c.charCodeAt(0)),

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

  // User:
  user: {
    id: "1098237235409872"
    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.
  parameters: [
    {
      type: "public-key",
      algorithm: "ES256",
    },
    {
      type: "public-key",
      algorithm: "RS256",
    },
  ],

  timeout: 60000,  // 1 minute
```

```
var options = { timeout: 60000,  // 1 minute
                excludeList: [],      // No excludeList
                extensions: {"webauthn.location": true}  // Include location inf
ormation
                                             // in attestation
};

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

11.2. Authentication

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

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

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

```
var webauthnAPI = navigator.authentication;

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

var challenge = new TextEncoder().encode("climb a mountain");
var options = {
                timeout = 60000,  // 1 minute
                allowList: [{ type: "ScopedCred" }]
```



```
  excludeList: [], // No excludeList
  extensions: {"webauthn.location": 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
.
  });
```

11.2. Authentication

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

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

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

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

var options = {
                challenge: new TextEncoder().encode("climb a mountain"),
                timeout: 60000,  // 1 minute
                allowList: [{ type: "public-key" }]
```

```
            };

webauthnAPI.getAssertion(challenge, 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.

```
var webauthnAPI = navigator.authentication;

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

var encoder = new TextEncoder();
var challenge = encoder.encode("climb a mountain");
var acceptableCredential1 = {
    type: "ScopedCred",
    id: encoder.encode("!!!!!!!hi there!!!!!!!\n")
};
var acceptableCredential2 = {
    type: "ScopedCred",
    id: encoder.encode("roses are red, violets are blue\n")
};

var options = {

                timeout: 60000,  // 1 minute
                allowList: [acceptableCredential1, acceptableCredential2];
                extensions: { 'webauthn.txauth.simple':
                   "Wave your hands in the air like you just don't care" };
            };

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

11.3. Decommissioning

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

* Possibility #1 -- user reports the credential as lost.
    + User goes to server.example.net, authenticates and follows a link to report a lost/stolen device.
    + Server returns a page showing the list of registered credentials with friendly names as configured during registration.
    + User selects a credential and the server deletes it from its database.
    + In future, the Relying Party script does not specify this credential in any list of acceptable credentials, and assertions signed by this credential are rejected.
* Possibility #2 -- server deregisters the credential due to inactivity.
    + Server deletes credential from its database during maintenance activity.
    + In 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.


```
            };

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

11.3. Decommissioning

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

* Possibility #1 -- user reports the credential as lost.
    + User goes to server.example.net, authenticates and follows a link to report a lost/stolen device.
    + Server returns a page showing the list of registered credentials with friendly names as configured during registration.
    + User selects a credential and the server deletes it from its database.
    + In future, the Relying Party script does not specify this credential in any list of acceptable credentials, and assertions signed by this credential are rejected.
* Possibility #2 -- server deregisters the credential due to inactivity.
    + Server deletes credential from its database during maintenance activity.
    + In 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.

12. Acknowledgements

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

Index

Terms defined by this specification

+ From this point on, this credential will not appear in any selection prompts, and no assertions can be generated with it.
+ Sometime later, the server deregisters this credential due to inactivity.

12. Acknowledgements

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

Index

Terms defined by this specification

* signature
  + attribute for AuthenticationAssertion, in 4.6
  + dfn for AuthenticationAssertion, in 4.6
* timeout
  + dict-member for ScopedCredentialOptions, in 4.5
  + dfn for ScopedCredentialOptions, in 4.5
  + dict-member for AssertionOptions, in 4.7
  + dfn for AssertionOptions, in 4.7
* tokenBinding
  + dict-member for ClientData, in 4.9.1
  + dfn for ClientData, in 4.9.1
* Transport, in 4.9.5
* transports, in 4.9.4
* type
  + dict-member for ScopedCredentialParameters, in 4.4
  + dfn for ScopedCredentialParameters, in 4.4
  + attribute for ScopedCredential, in 4.9.3
  + dfn for ScopedCredential, in 4.9.3
  + dict-member for ScopedCredentialDescriptor, in 4.9.4
  + dfn for ScopedCredentialDescriptor, in 4.9.4
* "usb", in 4.9.5
* usb
  + enum-value for Transport, in 4.9.5
  + dfn for Transport, in 4.9.5
* User Consent, in 3
* User Verification, in 3
* WebAuthentication, in 4.1
* Web Authentication API, in 4
* WebAuthn Client, in 3

Terms defined by reference

* [WHATWG-ENCODING] defines the following terms:
  + utf-8 encoded
* [HTML] defines the following terms:
  + Navigator
* [HTML51] defines the following terms:
  + current settings object
  + navigator
  + opaque origin
  + origin
  + relaxing the same-origin restriction

* signature, in 4.2.2
* Test of User Presence, in 3
* timeout
  + dict-member for MakeCredentialOptions, in 4.5
  + dict-member for PublicKeyCredentialRequestOptions, in 4.6
* tokenBinding, in 4.8.1
* Transport, in 4.8.4
* transports, in 4.8.3
* TUP, in 3
* [[type]], in 4.1
* type
  + dict-member for PublicKeyCredentialParameters, in 4.3
  + dict-member for PublicKeyCredentialDescriptor, in 4.8.3
* usb, in 4.8.4
* user, in 4.5
* User Consent, in 3
* User Verification, in 3
* User Verified, in 3
* Web Authentication API, in 4
* WebAuthn Client, in 3

Terms defined by reference

* [CREDENTIAL-MANAGEMENT-1] defines the following terms:
  + CredentialCreationOptions
  + create()
* [ECMAScript] defines the following terms:
  + %arraybuffer%
  + internal slot
  + stringify
* [ENCODING] defines the following terms:
  + utf-8 encode
* [HTML] defines the following terms:
  + ascii serialization of an origin
  + dom manipulation task source
  + effective domain
  + global object
  + in parallel
  + is a registrable domain suffix of or is equal to
  + is not a registrable domain suffix of and is not equal to
  + origin
  + relevant settings object
  + task
  + task source
  + unicode serialization of an origin
* [HTML52] defines the following terms:
  + opaque origin
  + origin
* [INFRA] defines the following terms:
  + append (for list)
  + append (for set)
  + continue
  + for each (for list)
  + for each (for map)
  + is empty
  + is not empty
  + item
  + list
  + map
  + ordered set
  + remove

* [WebCryptoAPI] defines the following terms:
  + normalizing an algorithm
* [WebIDL] defines the following terms:
  + BufferSource
  + SecureContext
  + present

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

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

  + set
* [secure-contexts] defines the following terms:
  + secure context
* [TokenBinding] defines the following terms:
  + token binding
  + token binding id
* [URL] defines the following terms:
  + url serializer
* [webappsec-credential-management-1] defines the following terms:
  + Credential
  + CredentialsContainer
  + [[CollectFromCredentialStore]](options)
  + [[Store]](credential)
  + [[discovery]]
  + [[type]]
  + get()
  + id
  + remote
  + type
* [WebCryptoAPI] defines the following terms:
  + AlgorithmIdentifier
  + normalizing an algorithm
  + recognized algorithm name
* [WebIDL] defines the following terms:
  + ArrayBuffer
  + BufferSource
  + ConstraintError
  + DOMException
  + DOMString
  + NotAllowedError
  + NotFoundError
  + NotSupportedError
  + Promise
  + SecureContext
  + SecurityError
  + TypeError
  + USVString
  + boolean
  + dictionary
  + interface object
  + present
  + simple exception
  + unsigned long

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. 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/

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

[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

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

[HTML]
Anne van Kesteren; et al. HTML Standard. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[HTML51]
Steve Faulkner; et al. HTML 5.1. 1 November 2016. REC. URL: https://www.w3.org/TR/html51/

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

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

[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

[RFC7518]
M. Jones. JSON Web Algorithms (JWA). May 2015. Proposed Standard. URL: https://tools.ietf.org/html/rfc7518

[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. URL: https://www.w3.org/TR/html52/

[INFRA]
Anne van Kesteren; Domenic Denicola. Infra Standard. Living Standard. URL: https://infra.spec.whatwg.org/

[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

[RFC7518]
M. Jones. JSON Web Algorithms (JWA). May 2015. Proposed Standard. URL: https://tools.ietf.org/html/rfc7518

[SECURE-CONTEXTS]
Mike West. Secure Contexts. 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/

[WebCryptoAPI]
Mark Watson. Web Cryptography API. 15 December 2016. PR. URL: https://www.w3.org/TR/WebCryptoAPI/

[WebIDL]
Cameron McCormack; Boris Zbarsky; Tobie Langel. Web IDL. 15 December 2016. WD. URL: https://www.w3.org/TR/WebIDL-1/

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

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

Informative References

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

[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-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

[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

[RFC4949]

[WEBAPPSEC-CREDENTIAL-MANAGEMENT-1]
Credential Management Level 1 URL: https://w3c.github.io/webappsec-credential-management/

[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. URL: https://www.w3.org/TR/WebCryptoAPI/

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

[WebIDL-1]
Cameron McCormack. WebIDL Level 1. 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

[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-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

[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

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

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

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

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

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

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

[TPMv2-Part2]
Trusted Platform Module Library, Part 2: Structures. URL: http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-2-Structures-01.16-1.pdf

[TPMv2-Part3]
Trusted Platform Module Library, Part 3: Commands. URL: http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-3-Commands-01.16-1.pdf

[UAFProtocol]
R. Lindemann; et al. FIDO UAF Protocol Specification v1.0. FIDO Alliance Proposed Standard. URL: https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-protocol-v1.0-ps-20141208.html

[WebAuthn-Registries]
Jeff Hodges. Registries for Web Authentication (WebAuthn). June 2016. 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#46923110554855074732

IDL Index

```
partial interface Navigator {
    readonly attribute WebAuthentication authentication;
```


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

[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

[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]
Trusted Platform Module Library, Part 1: Architecture. URL: http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-1-Architecture-01.38.pdf

[TPMv2-Part2]
Trusted Platform Module Library, Part 2: Structures. URL: http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-2-Structures-01.38.pdf

[TPMv2-Part3]
Trusted Platform Module Library, Part 3: Commands. URL: http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-3-Commands-01.38.pdf

[UAFProtocol]
R. Lindemann; et al. FIDO UAF Protocol Specification v1.0. FIDO Alliance Proposed Standard. URL: https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-protocol-v1.0-ps-20141208.html

IDL Index

```
[SecureContext]
interface PublicKeyCredential : Credential {
```

```
};

[SecureContext]
interface WebAuthentication {
    Promise<ScopedCredentialInfo> makeCredential(
        Account                    accountInformation,
        sequence<ScopedCredentialParameters> cryptoParameters,
        BufferSource               attestationChallenge,
        optional ScopedCredentialOptions     options
    );

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

[SecureContext]
interface ScopedCredentialInfo {
    readonly    attribute ArrayBuffer   clientDataJSON;

    readonly    attribute ArrayBuffer   attestationObject;
};

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

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

dictionary ScopedCredentialOptions {

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



```
    readonly attribute ArrayBuffer          rawId;
    readonly attribute AuthenticatorResponse response;
    readonly attribute AuthenticationExtensions clientExtensionResults;
};

partial dictionary CredentialRequestOptions {
    PublicKeyCredentialRequestOptions? publicKey;
};

partial dictionary CredentialCreationOptions {
    MakeCredentialOptions? publicKey;

};

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

[SecureContext]
interface AuthenticatorAttestationResponse : AuthenticatorResponse {
    readonly attribute ArrayBuffer attestationObject;
};

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

};

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

dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
    DOMString displayName;
};

dictionary MakeCredentialOptions {
    required PublicKeyCredentialEntity rp;
    required PublicKeyCredentialUserEntity user;

    required BufferSource                      challenge;
    required sequence<PublicKeyCredentialParameters> parameters;

    unsigned long                    timeout;
    sequence<PublicKeyCredentialDescriptor> excludeList;
    AuthenticatorSelectionCriteria       authenticatorSelection;
    AuthenticationExtensions           extensions;
};

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

dictionary AuthenticatorSelectionCriteria {
    Attachment    attachment;
    boolean       requireResidentKey = false;
```

```
enum Attachment {
    "platform",
    "cross-platform"
};

[SecureContext]
interface AuthenticationAssertion {
    readonly attribute ScopedCredential  credential;
    readonly attribute ArrayBuffer       clientDataJSON;
    readonly attribute ArrayBuffer       authenticatorData;
    readonly attribute ArrayBuffer       signature;
};

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

dictionary AuthenticationExtensions {
};

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

enum ScopedCredentialType {
    "ScopedCred"
};

[SecureContext]
interface ScopedCredential {
    readonly attribute ScopedCredentialType type;
    readonly attribute ArrayBuffer          id;
};

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

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

typedef sequence<AAGUID> AuthenticatorSelectionList;

typedef BufferSource AAGUID;
```

#domexceptionReferenced in:
 * 4.1.1. Create a new credential - makeCredential() method (2) (3)
 * 4.1.2. Use an existing credential - getAssertion() method (2) (3)

#dictdef-algorithmidentifierReferenced in:
 * 4.4. Parameters for Credential Generation (dictionary ScopedCredentialParameters)
 * 4.9.1. Client data used in WebAuthn signatures (dictionary ClientData)
 * 4.9.6. Cryptographic Algorithm Identifier (type


```
};

enum Attachment {
    "platform",
    "cross-platform"
};

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

typedef record<DOMString, any> AuthenticationExtensions;

dictionary CollectedClientData {
    required DOMString          challenge;
    required DOMString          origin;
    required DOMString          hashAlg;
    DOMString                   tokenBinding;
    AuthenticationExtensions    clientExtensions;
    AuthenticationExtensions    authenticatorExtensions;
};

enum PublicKeyCredentialType {
    "public-key"
};

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

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

typedef sequence<AAGUID> AuthenticatorSelectionList;

typedef BufferSource AAGUID;
```

#base64url-encodingReferenced in:
 * 4.1. PublicKeyCredential Interface
 * 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method (2)
 * 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method (2)
 * 6.2. Verifying an authentication assertion

AlgorithmIdentifier)

#promisesReferenced in:
* 4.1.1. Create a new credential - makeCredential() method
* 4.1.2. Use an existing credential - getAssertion() method

#ascii-case-insensitive-matchReferenced in:
* 6.1. Registering a new credential

#attestation-objectsReferenced in:
* 4. Web Authentication API
* 4.2. Information about Scoped Credential (interface ScopedCredentialInfo)
* 4.5. Additional options for Credential Generation (dictionary ScopedCredentialOptions)
* 5.2.1. The authenticatorMakeCredential operation (2)
* 5.3. Credential Attestation (2)
* 5.3.1. Attestation data
* 5.3.4. Generating an Attestation Object (2) (3) (4)
* 6.1. Registering a new credential

#attestation-certificateReferenced in:
* 3. Terminology (2)
* 7.3.1. TPM attestation statement certificate requirements

#attestation-key-pairReferenced in:
* 3. Terminology (2)

#authenticationReferenced in:
* 1. Introduction (2)
* 3. Terminology (2) (3)

#authentication-assertionReferenced in:
* 1. Introduction
* 3. Terminology (2) (3)

#authenticatorReferenced in:
* 1. Introduction (2) (3) (4)
* 1.1. Use Cases
* 2. Conformance
* 3. Terminology (2) (3) (4) (5) (6) (7) (8) (9)

#cborReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method
* 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
* 5.1. Authenticator data (2)
* 8. WebAuthn Extensions (2) (3)
* 8.2. Defining extensions (2)
* 8.3. Extending request parameters
* 8.4. Client extension processing (2)
* 8.5. Authenticator extension processing (2) (3) (4) (5)

#attestation-objectsReferenced in:
* 4. Web Authentication API
* 4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse) (2)
* 4.5. Options for Credential Creation (dictionary MakeCredentialOptions) (2)
* 5.2.1. The authenticatorMakeCredential operation (2)
* 5.3. Credential Attestation (2)
* 5.3.1. Attestation data
* 5.3.4. Generating an Attestation Object (2) (3) (4)
* 6.1. Registering a new credential

#attestation-certificateReferenced in:
* 3. Terminology (2)
* 7.3.1. TPM attestation statement certificate requirements

#attestation-key-pairReferenced in:
* 3. Terminology (2)

#authenticationReferenced in:
* 1. Introduction (2)
* 3. Terminology (2) (3) (4) (5)
* 6.2. Verifying an authentication assertion

#authentication-assertionReferenced in:
* 1. Introduction
* 3. Terminology (2) (3)
* 4.1. PublicKeyCredential Interface
* 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
* 4.6. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)
* 8. WebAuthn Extensions

#authenticatorReferenced in:
* 1. Introduction (2) (3) (4)
* 1.1. Use Cases
* 2. Conformance
* 3. Terminology (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12)
* 4.1. PublicKeyCredential Interface
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method
* 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
* 4.2. Authenticator Responses (interface AuthenticatorResponse)
* 4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse) (2)
* 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
* 4.5.3. Credential Attachment enumeration (enum Attachment)
* 4.6. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)

* 5. WebAuthn Authenticator model
* 5.1. Authenticator data
* 5.3. Credential Attestation
* 5.3.5.1. Privacy
* 5.3.5.2. Attestation Certificate and Attestation Certificate CA Compromise
* 7.2. Packed Attestation Statement Format
* 7.4. Android Key Attestation Statement Format
* 7.5. Android SafetyNet Attestation Statement Format
* 9.4. SupportedExtensions Extension
* 9.5. User Verification Index (UVI) Extension
* 9.6. Location Extension (2) (3) (4)
* 9.7. User Verification Mode (UVM) Extension
* 11. Sample scenarios

#authorization-gestureReferenced in:
* 1.1.1. Registration
* 1.1.2. Authentication
* 1.1.3. Other use cases and configurations

#ceremonyReferenced in:
* 1. Introduction
* 3. Terminology (2)

#conforming-user-agentReferenced in:
* 1. Introduction
* 2. Conformance (2) (3)
* 3. Terminology (2)

#credential-public-keyReferenced in:
* 3. Terminology
* 4.2. Information about Scoped Credential (interface ScopedCredentialInfo)
* 5.1. Authenticator data
* 7.4. Android Key Attestation Statement Format
* 11.1. Registration (2)

#credential-key-pairReferenced in:
* 3. Terminology (2)

#registrationReferenced in:
* 1. Introduction (2)
* 3. Terminology (2) (3) (4) (5) (6)

#relying-partyReferenced in:
* 1. Introduction (2) (3) (4)
* 3. Terminology (2) (3) (4) (5) (6) (7) (8)
* 4.2. Information about Scoped Credential (interface ScopedCredentialInfo)
* 4.3. User Account Information (dictionary Account)
* 4.5. Additional options for Credential Generation (dictionary

* 5. WebAuthn Authenticator model
* 5.1. Authenticator data
* 5.3. Credential Attestation
* 5.3.5.1. Privacy
* 5.3.5.2. Attestation Certificate and Attestation Certificate CA Compromise
* 6.1. Registering a new credential
* 7.2. Packed Attestation Statement Format
* 7.4. Android Key Attestation Statement Format
* 7.5. Android SafetyNet Attestation Statement Format
* 9.5. Supported Extensions Extension (exts)
* 9.6. User Verification Index Extension (uvi)
* 9.7. Location Extension (loc) (2) (3) (4)
* 9.8. User Verification Method Extension (uvm)
* 11. Sample scenarios

#authorization-gestureReferenced in:
* 1.1.1. Registration
* 1.1.2. Authentication
* 1.1.3. Other use cases and configurations
* 3. Terminology (2) (3) (4) (5) (6)

#biometric-recognitionReferenced in:
* 3. Terminology (2)

#ceremonyReferenced in:
* 1. Introduction
* 3. Terminology (2) (3) (4) (5) (6) (7)
* 6.1. Registering a new credential
* 6.2. Verifying an authentication assertion

#clientReferenced in:
* 3. Terminology

#conforming-user-agentReferenced in:
* 1. Introduction
* 2. Conformance (2) (3)
* 3. Terminology (2)

#credential-public-keyReferenced in:
* 3. Terminology
* 4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)
* 5.1. Authenticator data
* 7.4. Android Key Attestation Statement Format
* 11.1. Registration (2)

#credential-key-pairReferenced in:
* 3. Terminology (2)
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method

#credential-private-keyReferenced in:
* 3. Terminology (2)
* 4.1. PublicKeyCredential Interface
* 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)

#registrationReferenced in:
* 1. Introduction (2)
* 3. Terminology (2) (3) (4) (5) (6) (7)
* 6.1. Registering a new credential

#relying-partyReferenced in:
* 1. Introduction (2) (3) (4) (5)
* 1.1.3. Other use cases and configurations
* 3. Terminology (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13) (14)
* 4. Web Authentication API (2) (3)
* 4.2.1. Information about Public Key Credential (interface

ScopedCredentialOptions)
* 5. WebAuthn Authenticator model
* 5.3. Credential Attestation
* 6. Relying Party Operations
* 8.3. Extending request parameters
* 8.4. Extending client processing
* 8.6. Example extension
* 11.2. Authentication
* 11.3. Decommissioning

#relying-party-identifierReferenced in:
* 5. WebAuthn Authenticator model

#rp-idReferenced in:
* 3. Terminology (2)
* 5. WebAuthn Authenticator model

#scoped-credentialReferenced in:
* 1. Introduction (2) (3) (4) (5)
* 3. Terminology (2) (3) (4) (5) (6)

AuthenticatorAttestationResponse)
* 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
* 4.5. Options for Credential Creation (dictionary MakeCredentialOptions) (2) (3) (4) (5)
* 4.5.1. Entity Description (2) (3) (4) (5)
* 4.5.2. Authenticator Selection Criteria (2)
* 4.5.3. Credential Attachment enumeration (enum Attachment) (2)
* 4.8.1. Client data used in WebAuthn signatures (dictionary CollectedClientData) (2) (3) (4)
* 4.8.4. Credential Transport enumeration (enum ExternalTransport) (2)
* 5. WebAuthn Authenticator model (2)
* 5.1. Authenticator data (2)
* 5.2.1. The authenticatorMakeCredential operation (2) (3) (4)
* 5.2.2. The authenticatorGetAssertion operation (2) (3)
* 5.3. Credential Attestation (2) (3)
* 5.3.5.1. Privacy
* 5.3.5.2. Attestation Certificate and Attestation Certificate CA Compromise (2) (3) (4) (5)
* 6. Relying Party Operations (2) (3) (4)
* 6.1. Registering a new credential (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13)
* 6.2. Verifying an authentication assertion (2) (3) (4)
* 7.4. Android Key Attestation Statement Format
* 8. WebAuthn Extensions (2) (3)
* 8.2. Defining extensions (2)
* 8.3. Extending request parameters (2) (3) (4)
* 8.6. Example Extension (2) (3)
* 9.2. Simple Transaction Authorization Extension (txAuthSimple)
* 9.4. Authenticator Selection Extension (authnSel) (2)
* 9.5. Supported Extensions Extension (exts) (2)
* 9.6. User Verification Index Extension (uvi)
* 9.7. Location Extension (loc)
* 10.2. WebAuthn Extension Identifier Registrations
* 11.1. Registration (2) (3) (4)
* 11.2. Authentication (2) (3) (4) (5)
* 11.3. Decommissioning (2)

#relying-party-identifierReferenced in:
* 4.5. Options for Credential Creation (dictionary MakeCredentialOptions)
* 4.6. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)
* 5. WebAuthn Authenticator model

#rp-idReferenced in:
* 3. Terminology (2)
* 5. WebAuthn Authenticator model

#public-key-credentialReferenced in:
* 1. Introduction (2) (3) (4) (5)
* 3. Terminology (2) (3) (4) (5) (6)
* 4. Web Authentication API
* 4.1. PublicKeyCredential Interface
* 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
* 4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)
* 4.5.2. Authenticator Selection Criteria
* 4.6. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)
* 4.8. Supporting Data Structures
* 5. WebAuthn Authenticator model (2) (3)
* 5.3.3. Attestation Types
* 5.3.5.2. Attestation Certificate and Attestation Certificate CA Compromise (2)
* 6.1. Registering a new credential
* 8. WebAuthn Extensions (2)
* 11. Sample scenarios

#user-consentReferenced in:
 * 1. Introduction

#user-verificationReferenced in:
 * 1. Introduction
 * 3. Terminology (2) (3) (4) (5)

#webauthn-clientReferenced in:
 * 3. Terminology (2)

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

#webauthenticationReferenced in:
 * 4. Web Authentication API
 * 4.1. WebAuthentication Interface

#dom-webauthentication-makecredentialReferenced in:
 * 1. Introduction
 * 4.9.4. Credential Descriptor (dictionary ScopedCredentialDescriptor)
 * 6. Relying Party Operations
 * 6.1. Registering a new credential (2)
 * 8. WebAuthn Extensions (2)
 * 8.2. Defining extensions
 * 8.3. Extending request parameters (2)
 * 9.3. Authenticator Selection Extension

#dom-webauthentication-makecredential-accountinformation-cryptoparameters-attestationchallenge-options-accountinformationReferenced in:
 * 4.1.1. Create a new credential - makeCredential() method (2) (3)

#test-of-user-presenceReferenced in:
 * 3. Terminology (2)
 * 5.1. Authenticator data
 * 9.2. Simple Transaction Authorization Extension (txAuthSimple)
 * 9.3. Generic Transaction Authorization Extension (txAuthGeneric)

#tupReferenced in:
 * 3. Terminology

#client-side-resident-credential-private-keyReferenced in:
 * 3. Terminology (2)
 * 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method
 * 4.5.2. Authenticator Selection Criteria (2)
 * 5.2.1. The authenticatorMakeCredential operation

#user-consentReferenced in:
 * 1. Introduction
 * 3. Terminology (2)
 * 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method
 * 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)

#user-verificationReferenced in:
 * 1. Introduction
 * 3. Terminology (2) (3) (4) (5) (6) (7)
 * 9.2. Simple Transaction Authorization Extension (txAuthSimple)
 * 9.3. Generic Transaction Authorization Extension (txAuthGeneric)

#webauthn-clientReferenced in:
 * 3. Terminology (2)

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

#publickeycredentialReferenced in:

 * 1. Introduction
 * 4.1. PublicKeyCredential Interface (2) (3) (4) (5) (6) (7) (8)
 * 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method (2) (3) (4) (5) (6)
 * 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method (2)
 * 4.8.3. Credential Descriptor (dictionary PublicKeyCredentialDescriptor)
 * 5.2.1. The authenticatorMakeCredential operation (2)
 * 6. Relying Party Operations
 * 6.2. Verifying an authentication assertion

#dom-publickeycredential-rawidReferenced in:
 * 4.1. PublicKeyCredential Interface
 * 6.2. Verifying an authentication assertion

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

* 6.1. Registering a new credential

#dom-webauthentication-makecredential-accountinformation-cryptoparameters-attestationchallenge-options-cryptoparametersReferenced in:
* 4.1.1. Create a new credential - makeCredential() method (2) (3) (4) (5)

#dom-webauthentication-makecredential-accountinformation-cryptoparameters-attestationchallenge-options-attestationchallengeReferenced in:
* 4.1.1. Create a new credential - makeCredential() method

#dom-webauthentication-makecredential-accountinformation-cryptoparameters-attestationchallenge-options-optionsReferenced in:
* 4.1.1. Create a new credential - makeCredential() method (2) (3) (4) (5) (6) (7)

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

* 6.2. Verifying an authentication assertion

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

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

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

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

#dom-credentialcreationoptions-publickeyReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method

#dom-publickeycredential-create-slotReferenced in:
* 4.1. PublicKeyCredential Interface

#dom-publickeycredential-create-options-optionsReferenced in:
* 6.1. Registering a new credential

#dom-publickeycredential-discoverfromexternalsource-slotReferenced in:
* 4.1. PublicKeyCredential Interface

#authenticatorresponseReferenced in:
* 4.1. PublicKeyCredential Interface (2)
* 4.2. Authenticator Responses (interface AuthenticatorResponse) (2)
* 4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse) (2)
* 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse) (2)

#dom-authenticatorresponse-clientdatajsonReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method (2)
* 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method (2)
* 4.2. Authenticator Responses (interface AuthenticatorResponse)
* 4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)
* 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
* 6.1. Registering a new credential (2)
* 6.2. Verifying an authentication assertion

#authenticatorattestationresponseReferenced in:
* 4.1. PublicKeyCredential Interface
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method (2)
* 4.2.1. Information about Public Key Credential (interface

* 3. Terminology
* 4.1.1. Create a new credential - makeCredential() method (2)
* 4.8. Authentication Assertion Extensions (dictionary AuthenticationExtensions)
* 4.9.4. Credential Descriptor (dictionary ScopedCredentialDescriptor)
* 6. Relying Party Operations

* 6.2. Verifying an authentication assertion
* 8. WebAuthn Extensions (2)
* 8.2. Defining extensions
* 8.3. Extending request parameters (2)
* 9.1. FIDO AppId

#dom-webauthentication-getassertion-assertionchallenge-options-assertionchallengeReferenced in:
* 4.1.2. Use an existing credential - getAssertion() method

#dom-webauthentication-getassertion-assertionchallenge-options-optionsReferenced in:
* 4.1.2. Use an existing credential - getAssertion() method (2) (3) (4) (5)

#scopedcredentialinfoReferenced in:
* 4.1. WebAuthentication Interface
* 4.1.1. Create a new credential - makeCredential() method (2)
* 4.2. Information about Scoped Credential (interface ScopedCredentialInfo)
* 4.9.1. Client data used in WebAuthn signatures (dictionary ClientData)
* 6. Relying Party Operations
* 6.1. Registering a new credential (2) (3)

#dom-scopedcredentialinfo-clientdatajsonReferenced in:
* 4.1.1. Create a new credential - makeCredential() method (2)
* 6.1. Registering a new credential (2)

#dom-scopedcredentialinfo-attestationobjectReferenced in:
* 6.1. Registering a new credential

#dictdef-accountReferenced in:
* 4.1. WebAuthentication Interface

AuthenticatorAttestationResponse) (2)
* 6. Relying Party Operations
* 6.1. Registering a new credential (2) (3)

#dom-authenticatorattestationresponse-attestationobjectReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method
* 4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)
* 6.1. Registering a new credential

#authenticatorassertionresponseReferenced in:
* 3. Terminology
* 4.1. PublicKeyCredential Interface
* 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
* 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse) (2)
* 6. Relying Party Operations

#dom-authenticatorassertionresponse-authenticatordataReferenced in:
* 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method (2)
* 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
* 6.2. Verifying an authentication assertion

#dom-authenticatorassertionresponse-signatureReferenced in:
* 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method (2)
* 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
* 6.2. Verifying an authentication assertion

#dictdef-publickeycredentialparametersReferenced in:
* 4.3. Parameters for Credential Generation (dictionary PublicKeyCredentialParameters)
* 4.5. Options for Credential Creation (dictionary MakeCredentialOptions) (2)

#dom-publickeycredentialparameters-typeReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method (2)
* 4.3. Parameters for Credential Generation (dictionary PublicKeyCredentialParameters)

#dom-publickeycredentialparameters-algorithmReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method
* 4.3. Parameters for Credential Generation (dictionary PublicKeyCredentialParameters)

#dictdef-publickeycredentialuserentityReferenced in:
* 4.4. User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity)
* 4.5. Options for Credential Creation (dictionary MakeCredentialOptions) (2)

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

* 4.3. User Account Information (dictionary Account)
* 5.2.1. The authenticatorMakeCredential operation

#dom-account-idReferenced in:
* 4.1.1. Create a new credential - makeCredential() method
* 4.3. User Account Information (dictionary Account)
* 5.2.1. The authenticatorMakeCredential operation (2)

#dictdef-scopedcredentialparametersReferenced in:
* 4.1. WebAuthentication Interface
* 4.1.1. Create a new credential - makeCredential() method
* 4.4. Parameters for Credential Generation (dictionary ScopedCredentialParameters)

#dictdef-scopedcredentialoptionsReferenced in:
* 4.1. WebAuthentication Interface
* 4.5. Additional options for Credential Generation (dictionary ScopedCredentialOptions)

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

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

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

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

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

* 8.3. Extending request parameters

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

\[[Create]](options) method
* 4.4. User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity)
* 4.5. Options for Credential Creation (dictionary MakeCredentialOptions)

#dictdef-makecredentialoptionsReferenced in:
* 4.1.2. CredentialCreationOptions Extension
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method
* 4.5. Options for Credential Creation (dictionary MakeCredentialOptions)

#dom-makecredentialoptions-rpReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method (2) (3) (4) (5)
* 4.5. Options for Credential Creation (dictionary MakeCredentialOptions)

#dom-makecredentialoptions-userReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method (2) (3) (4)
* 4.5. Options for Credential Creation (dictionary MakeCredentialOptions)
* 5.2.1. The authenticatorMakeCredential operation (2)
* 6.1. Registering a new credential

#dom-makecredentialoptions-challengeReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method
* 4.5. Options for Credential Creation (dictionary MakeCredentialOptions)

#dom-makecredentialoptions-parametersReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method (2)
* 4.5. Options for Credential Creation (dictionary MakeCredentialOptions)

#dom-makecredentialoptions-timeoutReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method (2)
* 4.5. Options for Credential Creation (dictionary MakeCredentialOptions)

#dom-makecredentialoptions-excludelistReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method
* 4.5. Options for Credential Creation (dictionary MakeCredentialOptions)

#dom-makecredentialoptions-authenticatorselectionReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method
* 4.5. Options for Credential Creation (dictionary MakeCredentialOptions)
* 5.2.1. The authenticatorMakeCredential operation

#dom-makecredentialoptions-extensionsReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method (2)
* 4.5. Options for Credential Creation (dictionary MakeCredentialOptions)
* 8.3. Extending request parameters

#dictdef-publickeycredentialentityReferenced in:
* 4.4. User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity)
* 4.5. Options for Credential Creation (dictionary MakeCredentialOptions) (2)

#attachment-platform-authenticatorsReferenced in:
* 4.5.1. Credential Attachment enumeration (enum Attachment) (2)

#attachment-roaming-authenticatorsReferenced in:
* 1.1.3. Other use cases and configurations
* 4.5.1. Credential Attachment enumeration (enum Attachment) (2)

#attachment-platform-attachmentReferenced in:
* 4.5.1. Credential Attachment enumeration (enum Attachment)

#attachment-cross-platform-attachedReferenced in:
* 4.5.1. Credential Attachment enumeration (enum Attachment) (2)

#authenticationassertionReferenced in:
* 3. Terminology
* 4.1. WebAuthentication Interface
* 4.1.2. Use an existing credential - getAssertion() method
* 4.6. Web Authentication Assertion (interface AuthenticationAssertion)
* 4.9.1. Client data used in WebAuthn signatures (dictionary ClientData)
* 6. Relying Party Operations
* 6.2. Verifying an authentication assertion (2) (3)

#dom-authenticationassertion-credentialReferenced in:
* 6.2. Verifying an authentication assertion

#dom-authenticationassertion-clientdatajsonReferenced in:
* 4.1.2. Use an existing credential - getAssertion() method (2)
* 6.2. Verifying an authentication assertion

#dom-authenticationassertion-authenticatordataReferenced in:
* 6.2. Verifying an authentication assertion

#dom-authenticationassertion-signatureReferenced in:
* 6.2. Verifying an authentication assertion

#dictdef-assertionoptionsReferenced in:
* 4.1. WebAuthentication Interface
* 4.7. Additional options for Assertion Generation (dictionary AssertionOptions)

* 4.5.1. Entity Description
* 5.2.1. The authenticatorMakeCredential operation (2)

#dom-publickeycredentialentity-idReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method (2) (3) (4)
* 4.5. Options for Credential Creation (dictionary MakeCredentialOptions) (2) (3)
* 4.5.1. Entity Description
* 5.2.1. The authenticatorMakeCredential operation (2)

#dom-publickeycredentialentity-nameReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method (2)
* 4.5. Options for Credential Creation (dictionary MakeCredentialOptions) (2)
* 4.5.1. Entity Description

#dom-publickeycredentialentity-iconReferenced in:
* 4.5.1. Entity Description

#dictdef-authenticatorselectioncriteriaReferenced in:
* 4.5. Options for Credential Creation (dictionary MakeCredentialOptions) (2)
* 4.5.2. Authenticator Selection Criteria

#dom-authenticatorselectioncriteria-attachmentReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method
* 4.5.2. Authenticator Selection Criteria

#dom-authenticatorselectioncriteria-requireresidentkeyReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method
* 4.5.2. Authenticator Selection Criteria

#enumdef-attachmentReferenced in:
* 4.5.2. Authenticator Selection Criteria (2)
* 4.5.3. Credential Attachment enumeration (enum Attachment) (2)

#platform-authenticatorsReferenced in:
* 4.5.3. Credential Attachment enumeration (enum Attachment) (2)

#roaming-authenticatorsReferenced in:
* 1.1.3. Other use cases and configurations
* 4.5.3. Credential Attachment enumeration (enum Attachment) (2)

#platform-attachmentReferenced in:
* 4.5.3. Credential Attachment enumeration (enum Attachment)

#cross-platform-attachedReferenced in:
* 4.5.3. Credential Attachment enumeration (enum Attachment) (2)

#dictdef-publickeycredentialrequestoptionsReferenced in:
* 4.1.1. CredentialRequestOptions Extension
* 4.6. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions) (2)
* 6.2. Verifying an authentication assertion

#dom-publickeycredentialrequestoptions-challengeReferenced in:
* 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
* 4.6. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions) (2)

#dom-assertionoptions-timeoutReferenced in:
 * 4.1.2. Use an existing credential - getAssertion() method (2)

#dom-assertionoptions-rpidReferenced in:
 * 4.1.2. Use an existing credential - getAssertion() method (2) (3)
 * 9.1. FIDO AppId

#dom-assertionoptions-allowlistReferenced in:
 * 4.1.1. Create a new credential - makeCredential() method (2)
 * 4.1.2. Use an existing credential - getAssertion() method (2) (3)

#dom-assertionoptions-extensionsReferenced in:
 * 4.1.2. Use an existing credential - getAssertion() method

#dictdef-authenticationextensionsReferenced in:
 * 4.5. Additional options for Credential Generation (dictionary ScopedCredentialOptions)
 * 4.7. Additional options for Assertion Generation (dictionary AssertionOptions)
 * 4.8. Authentication Assertion Extensions (dictionary AuthenticationExtensions)
 * 4.9.1. Client data used in WebAuthn signatures (dictionary ClientData)

#dictdef-clientdataReferenced in:
 * 4.1.1. Create a new credential - makeCredential() method
 * 4.1.2. Use an existing credential - getAssertion() method
 * 4.9.1. Client data used in WebAuthn signatures (dictionary ClientData) (2)
 * 5. WebAuthn Authenticator model
 * 5.1. Authenticator data
 * 5.2.1. The authenticatorMakeCredential operation
 * 5.2.2. The authenticatorGetAssertion operation
 * 6.1. Registering a new credential
 * 6.2. Verifying an authentication assertion
 * 8. WebAuthn Extensions (2)
 * 8.2. Defining extensions (2)
 * 8.4. Extending client processing (2)

#dom-clientdata-challengeReferenced in:
 * 6.1. Registering a new credential
 * 6.2. Verifying an authentication assertion

#dom-publickeycredentialrequestoptions-timeoutReferenced in:
 * 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method (2)
 * 4.6. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)

#dom-publickeycredentialrequestoptions-rpidReferenced in:
 * 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method (2) (3)
 * 4.6. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)
 * 9.1. FIDO AppId Extension (appid)

#dom-publickeycredentialrequestoptions-allowlistReferenced in:
 * 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method (2) (3) (4)
 * 4.6. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)

#dom-publickeycredentialrequestoptions-extensionsReferenced in:
 * 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method (2)
 * 4.6. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)

#typedefdef-authenticationextensionsReferenced in:
 * 4.1. PublicKeyCredential Interface (2)
 * 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method
 * 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
 * 4.5. Options for Credential Creation (dictionary MakeCredentialOptions) (2)
 * 4.6. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions) (2)
 * 4.8.1. Client data used in WebAuthn signatures (dictionary CollectedClientData) (2)

#dictdef-collectedclientdataReferenced in:
 * 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method
 * 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
 * 4.8.1. Client data used in WebAuthn signatures (dictionary CollectedClientData) (2)

#client-dataReferenced in:
 * 4.2. Authenticator Responses (interface AuthenticatorResponse)
 * 5. WebAuthn Authenticator model (2) (3)
 * 5.1. Authenticator data (2)
 * 6.1. Registering a new credential
 * 6.2. Verifying an authentication assertion
 * 8. WebAuthn Extensions
 * 8.4. Client extension processing
 * 8.6. Example Extension

#dom-collectedclientdata-challengeReferenced in:
 * 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method
 * 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
 * 4.8.1. Client data used in WebAuthn signatures (dictionary CollectedClientData)
 * 6.1. Registering a new credential
 * 6.2. Verifying an authentication assertion

#dom-clientdata-originReferenced in:
* 6.1. Registering a new credential
* 6.2. Verifying an authentication assertion

#dom-clientdata-hashalgReferenced in:
* 4.1.1. Create a new credential - makeCredential() method
* 4.1.2. Use an existing credential - getAssertion() method
* 6.1. Registering a new credential
* 6.2. Verifying an authentication assertion

#dom-clientdata-tokenbindingReferenced in:
* 6.1. Registering a new credential
* 6.2. Verifying an authentication assertion

#dom-clientdata-extensionsReferenced in:
* 6.1. Registering a new credential
* 6.2. Verifying an authentication assertion
* 8.4. Extending client processing

#clientdata-hashalgReferenced in:
* 4.9.1. Client data used in WebAuthn signatures (dictionary ClientData)

#clientdata-clientdatajsonReferenced in:
* 4.2. Information about Scoped Credential (interface ScopedCredentialInfo)
* 4.9.1. Client data used in WebAuthn signatures (dictionary ClientData)

#clientdata-clientdatahashReferenced in:
* 4.1.1. Create a new credential - makeCredential() method (2)
* 4.1.2. Use an existing credential - getAssertion() method (2)
* 4.2. Information about Scoped Credential (interface ScopedCredentialInfo)

#dom-collectedclientdata-originReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method
* 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
* 4.8.1. Client data used in WebAuthn signatures (dictionary CollectedClientData)
* 6.1. Registering a new credential
* 6.2. Verifying an authentication assertion

#dom-collectedclientdata-hashalgReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method
* 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
* 4.8.1. Client data used in WebAuthn signatures (dictionary CollectedClientData) (2)
* 6.1. Registering a new credential
* 6.2. Verifying an authentication assertion

#dom-collectedclientdata-tokenbindingReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method
* 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
* 4.8.1. Client data used in WebAuthn signatures (dictionary CollectedClientData)
* 6.1. Registering a new credential
* 6.2. Verifying an authentication assertion

#dom-collectedclientdata-clientextensionsReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method
* 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
* 4.8.1. Client data used in WebAuthn signatures (dictionary CollectedClientData)
* 6.1. Registering a new credential
* 6.2. Verifying an authentication assertion
* 8.4. Client extension processing

#dom-collectedclientdata-authenticatorextensionsReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method
* 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
* 4.8.1. Client data used in WebAuthn signatures (dictionary CollectedClientData)
* 6.1. Registering a new credential
* 6.2. Verifying an authentication assertion

#collectedclientdata-json-serialized-client-dataReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method
* 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
* 4.2. Authenticator Responses (interface AuthenticatorResponse)
* 4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse) (2)
* 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
* 4.8.1. Client data used in WebAuthn signatures (dictionary CollectedClientData)

#collectedclientdata-hash-of-the-serialized-client-dataReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method (2)
* 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method

* 4.9.1. Client data used in WebAuthn signatures (dictionary
  ClientData)
* 5. WebAuthn Authenticator model
* 5.2.1. The authenticatorMakeCredential operation (2)
* 5.2.2. The authenticatorGetAssertion operation (2) (3)
* 5.3.2. Attestation Statement Formats (2)
* 5.3.4. Generating an Attestation Object
* 6.1. Registering a new credential (2)
* 7.2. Packed Attestation Statement Format (2) (3) (4) (5) (6)
* 7.3. TPM Attestation Statement Format (2)
* 7.4. Android Key Attestation Statement Format (2)
* 7.5. Android SafetyNet Attestation Statement Format (2)
* 7.6. FIDO U2F Attestation Statement Format (2) (3) (4)

#enumdef-scopedcredentialtypeReferenced in:
* 4.1.1. Create a new credential - makeCredential() method
* 4.4. Parameters for Credential Generation (dictionary
  ScopedCredentialParameters)
* 4.9.2. Credential Type enumeration (enum ScopedCredentialType)
* 4.9.3. Unique Identifier for Credential (interface
  ScopedCredential) (2)
* 4.9.4. Credential Descriptor (dictionary
  ScopedCredentialDescriptor)
* 5.2.1. The authenticatorMakeCredential operation (2) (3)

#scopedcredentialReferenced in:
* 4.6. Web Authentication Assertion (interface
  AuthenticationAssertion)
* 4.9.3. Unique Identifier for Credential (interface
  ScopedCredential)
* 4.9.4. Credential Descriptor (dictionary
  ScopedCredentialDescriptor)
* 5.2.1. The authenticatorMakeCredential operation (2)

#dom-scopedcredential-idReferenced in:
* 6.2. Verifying an authentication assertion

#dictdef-scopedcredentialdescriptorReferenced in:
* 4.5. Additional options for Credential Generation (dictionary
  ScopedCredentialOptions)
* 4.7. Additional options for Assertion Generation (dictionary
  AssertionOptions)
* 4.9.4. Credential Descriptor (dictionary
  ScopedCredentialDescriptor)

#enumdef-transportReferenced in:
* 4.9.4. Credential Descriptor (dictionary
  ScopedCredentialDescriptor)

  (2)
* 4.2.1. Information about Public Key Credential (interface
  AuthenticatorAttestationResponse)
* 4.2.2. Web Authentication Assertion (interface
  AuthenticatorAssertionResponse)
* 4.8.1. Client data used in WebAuthn signatures (dictionary
  CollectedClientData)
* 5. WebAuthn Authenticator model
* 5.2.1. The authenticatorMakeCredential operation (2)
* 5.2.2. The authenticatorGetAssertion operation (2) (3)
* 5.3.2. Attestation Statement Formats (2)
* 5.3.4. Generating an Attestation Object
* 6.1. Registering a new credential
* 7.2. Packed Attestation Statement Format (2)
* 7.3. TPM Attestation Statement Format (2)
* 7.4. Android Key Attestation Statement Format (2)
* 7.5. Android SafetyNet Attestation Statement Format
* 7.6. FIDO U2F Attestation Statement Format (2)

#enumdef-publickeycredentialtypeReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's
  \[[Create]](options) method (2)
* 4.3. Parameters for Credential Generation (dictionary
  PublicKeyCredentialParameters)
* 4.8.2. Credential Type enumeration (enum PublicKeyCredentialType)
* 4.8.3. Credential Descriptor (dictionary
  PublicKeyCredentialDescriptor)
* 5.2.1. The authenticatorMakeCredential operation (2) (3)

#dom-publickeycredentialtype-public-keyReferenced in:
* 4.8.2. Credential Type enumeration (enum PublicKeyCredentialType)

#dictdef-publickeycredentialdescriptorReferenced in:
* 4.5. Options for Credential Creation (dictionary
  MakeCredentialOptions) (2)
* 4.6. Options for Assertion Generation (dictionary
  PublicKeyCredentialRequestOptions) (2) (3)
* 4.8.3. Credential Descriptor (dictionary
  PublicKeyCredentialDescriptor)

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

#dom-publickeycredentialdescriptor-typeReferenced in:
* 4.1.4. Use an existing credential -
  PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
* 4.8.3. Credential Descriptor (dictionary
  PublicKeyCredentialDescriptor)

#dom-publickeycredentialdescriptor-idReferenced in:
* 4.1.4. Use an existing credential -
  PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
* 4.8.3. Credential Descriptor (dictionary
  PublicKeyCredentialDescriptor)

#enumdef-transportReferenced in:
* 4.8.3. Credential Descriptor (dictionary
  PublicKeyCredentialDescriptor)

#authenticatordataReferenced in:
  * 5.1. Authenticator data (2)
  * 5.2.1. The authenticatorMakeCredential operation (2)
  * 5.2.2. The authenticatorGetAssertion operation (2) (3) (4)
  * 5.3.1. Attestation data
  * 5.3.2. Attestation Statement Formats (2)
  * 5.3.4. Generating an Attestation Object (2)
  * 5.3.5.3. Attestation Certificate Hierarchy
  * 7.2. Packed Attestation Statement Format (2) (3) (4) (5) (6) (7) (8)
  * 7.3. TPM Attestation Statement Format (2) (3) (4)
  * 7.4. Android Key Attestation Statement Format (2) (3) (4) (5)
  * 7.5. Android SafetyNet Attestation Statement Format (2)
  * 7.6. FIDO U2F Attestation Statement Format
  * 8. WebAuthn Extensions (2)
  * 8.2. Defining extensions (2)
  * 8.5. Extending authenticator processing (2)
  * 8.6. Example extension
  * 9.5. User Verification Index (UVI) Extension
  * 9.6. Location Extension
  * 9.7. User Verification Mode (UVM) Extension

#authenticatormakecredentialReferenced in:
  * 3. Terminology (2)
  * 4.1.1. Create a new credential - makeCredential() method
  * 5.2.3. The authenticatorCancel operation (2)
  * 8. WebAuthn Extensions
  * 8.2. Defining extensions

#authenticatorgetassertionReferenced in:
  * 3. Terminology (2)
  * 4.1.2. Use an existing credential - getAssertion() method
  * 5. WebAuthn Authenticator model
  * 5.1. Authenticator data
  * 5.2.3. The authenticatorCancel operation (2)
  * 8. WebAuthn Extensions
  * 8.2. Defining extensions

#authenticatorcancelReferenced in:
  * 4.1.1. Create a new credential - makeCredential() method (2) (3)
  * 4.1.2. Use an existing credential - getAssertion() method (2) (3)

#attestation-statement-formatReferenced in:
  * 3. Terminology
  * 4.2. Information about Scoped Credential (interface ScopedCredentialInfo)
  * 5.3.4. Generating an Attestation Object (2)

#attestation-typeReferenced in:

#dom-transport-usbReferenced in:
  * 4.8.4. Credential Transport enumeration (enum ExternalTransport)

#dom-transport-nfcReferenced in:
  * 4.8.4. Credential Transport enumeration (enum ExternalTransport)

#dom-transport-bleReferenced in:
  * 4.8.4. Credential Transport enumeration (enum ExternalTransport)

#authenticator-dataReferenced in:
  * 4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse) (2)
  * 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
  * 5. WebAuthn Authenticator model (2)
  * 5.1. Authenticator data (2) (3) (4) (5) (6) (7) (8)
  * 5.2.1. The authenticatorMakeCredential operation (2)
  * 5.2.2. The authenticatorGetAssertion operation (2) (3) (4)
  * 5.3. Credential Attestation (2)
  * 5.3.1. Attestation data
  * 5.3.2. Attestation Statement Formats (2)
  * 5.3.4. Generating an Attestation Object (2) (3)
  * 5.3.5.3. Attestation Certificate Hierarchy
  * 6.1. Registering a new credential (2)
  * 7.5. Android SafetyNet Attestation Statement Format
  * 8.5. Authenticator extension processing (2)
  * 8.6. Example Extension (2)
  * 9.6. User Verification Index Extension (uvi)
  * 9.7. Location Extension (loc)
  * 9.8. User Verification Method Extension (uvm)

#authenticatormakecredentialReferenced in:
  * 3. Terminology (2) (3)
  * 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method (2)
  * 5.2.3. The authenticatorCancel operation (2)
  * 8. WebAuthn Extensions
  * 8.2. Defining extensions

#authenticatorgetassertionReferenced in:
  * 3. Terminology (2) (3)
  * 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method (2) (3)
  * 5. WebAuthn Authenticator model
  * 5.1. Authenticator data
  * 5.2.3. The authenticatorCancel operation (2)
  * 8. WebAuthn Extensions
  * 8.2. Defining extensions

#authenticatorcancelReferenced in:
  * 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method (2) (3)
  * 4.1.4. Use an existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method (2) (3)

#attestation-statement-formatReferenced in:
  * 3. Terminology
  * 4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)
  * 5.3.4. Generating an Attestation Object (2)

#attestation-typeReferenced in:

* 3. Terminology

#basic-attestationReferenced in:
 * 5.3.5.1. Privacy

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

#privacy-caReferenced in:
 * 5.3.5.1. Privacy

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

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

#client-argumentReferenced in:

* 3. Terminology

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

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

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

#basic-attestationReferenced in:
 * 5.3.5.1. Privacy

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

#privacy-caReferenced in:
 * 5.3.5.1. Privacy

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

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

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

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

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

#registration-extensionReferenced in:
 * 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options) method
 * 8. WebAuthn Extensions (2) (3) (4) (5) (6)
 * 8.6. Example Extension

#typedefdef-aaguidReferenced in:
 * 9.3. Authenticator Selection Extension

 * 8. WebAuthn Extensions (2) (3) (4) (5)
 * 8.2. Defining extensions
 * 8.3. Extending request parameters (2) (3)
 * 8.4. Client extension processing
 * 8.5. Authenticator extension processing (2) (3)

#client-extension-processingReferenced in:
 * 4.1. PublicKeyCredential Interface
 * 4.1.3. Create a new credential - PublicKeyCredential's
 \[[Create]](options) method (2)
 * 4.1.4. Use an existing credential -
 PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
 (2)
 * 8. WebAuthn Extensions (2) (3) (4)
 * 8.2. Defining extensions

#client-extension-outputReferenced in:
 * 4.1. PublicKeyCredential Interface
 * 4.1.3. Create a new credential - PublicKeyCredential's
 \[[Create]](options) method (2)
 * 4.1.4. Use an existing credential -
 PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
 (2)
 * 8. WebAuthn Extensions (2) (3)
 * 8.2. Defining extensions (2) (3)
 * 8.4. Client extension processing (2) (3)
 * 8.6. Example Extension

#authenticator-extension-processingReferenced in:
 * 8. WebAuthn Extensions
 * 8.2. Defining extensions
 * 8.5. Authenticator extension processing

#authenticator-extension-outputReferenced in:
 * 5.1. Authenticator data
 * 8. WebAuthn Extensions (2) (3)
 * 8.2. Defining extensions (2) (3)
 * 8.4. Client extension processing
 * 8.5. Authenticator extension processing
 * 8.6. Example Extension
 * 9.5. Supported Extensions Extension (exts)
 * 9.6. User Verification Index Extension (uvi)
 * 9.7. Location Extension (loc)
 * 9.8. User Verification Method Extension (uvm)

#typedefdef-authenticatorselectionlistReferenced in:
 * 9.4. Authenticator Selection Extension (authnSel)

#typedefdef-aaguidReferenced in:
 * 9.4. Authenticator Selection Extension (authnSel)