THE_URL:file://localhost/Users/jehodges/documents/work/standards/W3C/WebAuthn/index-master-tr-598ac41-WD-06.html
THE_TITLE:Web Authentication: An API for accessing Public Key Credentials Level 1
^] Jump to Table of Contents-> Pop Out Sidebar

W3C

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

W3C Working Draft, 11 August 2017

This version:
https://www.w3.org/TR/2017/WD-webauthn-20170811/

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

Issue Tracking:
Github

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

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

Abstract

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

Status of this document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of

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

W3C

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

W3C Working Draft, 5 December 2017

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

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

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

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

Issue Tracking:
Github

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

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

Abstract

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

Status of this document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of

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

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

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

This document was produced by a group operating under the 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. 1 Introduction
    1. 1.1 Use Cases
        1. 1.1.1 Registration
        2. 1.1.2 Authentication
        3. 1.1.3 Other use cases and configurations
2. 2 Conformance
    1. 2.1 Dependencies
3. 3 Terminology
4. 4 Web Authentication API
    1. 4.1 PublicKeyCredential Interface
        1. 4.1.1 CredentialCreationOptions Extension
        2. 4.1.2 CredentialRequestOptions Extension
        3. 4.1.3 Create a new credential - PublicKeyCredential's [[Create]](options) method
        4. 4.1.4 Use an existing credential to make an assertion - PublicKeyCredential's [[DiscoverFromExternalSource]](options) method
        5. 4.1.5 Platform Authenticator Availability - PublicKeyCredential's isPlatformAuthenticatorAvailable() method
    2. 4.2 Authenticator Responses (interface AuthenticatorResponse)
        1. 4.2.1 Information about Public Key Credential (interface AuthenticatorAttestationResponse)
        2. 4.2.2 Web Authentication Assertion (interface AuthenticatorAssertionResponse)
    3. 4.3 Parameters for Credential Generation (dictionary PublicKeyCredentialParameters)
    4. 4.4 Options for Credential Creation (dictionary MakePublicKeyCredentialOptions)
        1. 4.4.1 Public Key Entity Description (dictionary PublicKeyCredentialEntity)
        2. 4.4.2 User Account Parameters for Credential Generation

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

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

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

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

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

Table of Contents

1. 1 Introduction
    1. 1.1 Use Cases
        1. 1.1.1 Registration
        2. 1.1.2 Authentication
        3. 1.1.3 Other use cases and configurations
2. 2 Conformance
    1. 2.1 User Agents
    2. 2.2 Authenticators
    3. 2.3 Relying Parties
3. 3 Dependencies
4. 4 Terminology
5. 5 Web Authentication API
    1. 5.1 PublicKeyCredential Interface
        1. 5.1.1 CredentialCreationOptions Extension
        2. 5.1.2 CredentialRequestOptions Extension
        3. 5.1.3 Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
        4. 5.1.4 Use an existing credential to make an assertion - PublicKeyCredential's [[Get]](options) method
            1. 5.1.4.1 PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method
        5. 5.1.5 Store an existing credential - PublicKeyCredential's [[Store]](credential, sameOriginWithAncestors) method
        6. 5.1.6 Availability of User-Verifying Platform Authenticator - PublicKeyCredential's isUserVerifyingPlatformAuthenticatorAvailable() method
    2. 5.2 Authenticator Responses (interface AuthenticatorResponse)
        1. 5.2.1 Information about Public Key Credential (interface AuthenticatorAttestationResponse)
        2. 5.2.2 Web Authentication Assertion (interface AuthenticatorAssertionResponse)
    3. 5.3 Parameters for Credential Generation (dictionary PublicKeyCredentialParameters)
    4. 5.4 Options for Credential Creation (dictionary MakePublicKeyCredentialOptions)
        1. 5.4.1 Public Key Entity Description (dictionary PublicKeyCredentialEntity)
        2. 5.4.2 RP Parameters for Credential Generation (dictionary

/Users/jehodges/Documents/work/standards/W3C/webauthn/index-master-tr-598ac41-WD-06.txt, Top line: 128

(dictionary PublicKeyCredentialUserEntity)
3. 4.4.3 Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria)
4. 4.4.4 Authenticator Attachment enumeration (enum AuthenticatorAttachment)
5. 4.5 Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)
6. 4.6 Authentication Extensions (typedef AuthenticationExtensions)
7. 4.7 Supporting Data Structures
1. 4.7.1 Client data used in WebAuthn signatures (dictionary CollectedClientData)
2. 4.7.2 Credential Type enumeration (enum PublicKeyCredentialType)
3. 4.7.3 Credential Descriptor (dictionary PublicKeyCredentialDescriptor)
4. 4.7.4 Authenticator Transport enumeration (enum AuthenticatorTransport)
5. 4.7.5 Cryptographic Algorithm Identifier (typedef COSEAlgorithmIdentifier)
5. 5 WebAuthn Authenticator model
1. 5.1 Authenticator data
2. 5.2 Authenticator operations
1. 5.2.1 The authenticatorMakeCredential operation
2. 5.2.2 The authenticatorGetAssertion operation
3. 5.2.3 The authenticatorCancel operation
3. 5.3 Attestation
1. 5.3.1 Attestation data
2. 5.3.2 Attestation Statement Formats
3. 5.3.3 Attestation Types
4. 5.3.4 Generating an Attestation Object
5. 5.3.5 Security Considerations
1. 5.3.5.1 Privacy
2. 5.3.5.2 Attestation Certificate and Attestation Certificate CA Compromise
3. 5.3.5.3 Attestation Certificate Hierarchy
6. 6 Relying Party Operations
1. 6.1 Registering a new credential
2. 6.2 Verifying an authentication assertion
7. 7 Defined Attestation Statement Formats
1. 7.1 Attestation Statement Format Identifiers
2. 7.2 Packed Attestation Statement Format
1. 7.2.1 Packed attestation statement certificate requirements
3. 7.3 TPM Attestation Statement Format
1. 7.3.1 TPM attestation statement certificate requirements
4. 7.4 Android Key Attestation Statement Format
5. 7.5 Android SafetyNet Attestation Statement Format
6. 7.6 FIDO U2F Attestation Statement Format
8. 8 WebAuthn Extensions
1. 8.1 Extension Identifiers
2. 8.2 Defining extensions
3. 8.3 Extending request parameters
4. 8.4 Client extension processing
5. 8.5 Authenticator extension processing
6. 8.6 Example Extension
9. 9 Defined Extensions
1. 9.1 FIDO AppId Extension (appid)
2. 9.2 Simple Transaction Authorization Extension (txAuthSimple)
3. 9.3 Generic Transaction Authorization Extension (txAuthGeneric)
4. 9.4 Authenticator Selection Extension (authnSel)

/Users/jehodges/Documents/work/standards/W3C/webauthn/index-master-tr-5e63e57-WD-07.txt, Top line: 138

PublicKeyCredentialRpEntity)
3. 5.4.3 User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity)
4. 5.4.4 Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria)
5. 5.4.5 Authenticator Attachment enumeration (enum AuthenticatorAttachment)
6. 5.4.6 Attestation Conveyance Preference enumeration (enum AttestationConveyancePreference)
5. 5.5 Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)
6. 5.6 Abort operations with AbortSignal
7. 5.7 Authentication Extensions (typedef AuthenticationExtensions)
8. 5.8 Supporting Data Structures
1. 5.8.1 Client data used in WebAuthn signatures (dictionary CollectedClientData)
2. 5.8.2 Credential Type enumeration (enum PublicKeyCredentialType)
3. 5.8.3 Credential Descriptor (dictionary PublicKeyCredentialDescriptor)
4. 5.8.4 Authenticator Transport enumeration (enum AuthenticatorTransport)
5. 5.8.5 Cryptographic Algorithm Identifier (typedef COSEAlgorithmIdentifier)
6. 5.8.6 User Verification Requirement enumeration (enum UserVerificationRequirement)
6. 6 WebAuthn Authenticator model
1. 6.1 Authenticator data
1. 6.1.1 Signature Counter Considerations
2. 6.2 Authenticator operations
1. 6.2.1 The authenticatorMakeCredential operation
2. 6.2.2 The authenticatorGetAssertion operation
3. 6.2.3 The authenticatorCancel operation
3. 6.3 Attestation
1. 6.3.1 Attested credential data
2. 6.3.2 Attestation Statement Formats
3. 6.3.3 Attestation Types
4. 6.3.4 Generating an Attestation Object
5. 6.3.5 Security Considerations
1. 6.3.5.1 Privacy
2. 6.3.5.2 Attestation Certificate and Attestation Certificate CA Compromise
3. 6.3.5.3 Attestation Certificate Hierarchy
7. 7 Relying Party Operations
1. 7.1 Registering a new credential
2. 7.2 Verifying an authentication assertion
8. 8 Defined Attestation Statement Formats
1. 8.1 Attestation Statement Format Identifiers
2. 8.2 Packed Attestation Statement Format
1. 8.2.1 Packed attestation statement certificate requirements
3. 8.3 TPM Attestation Statement Format
1. 8.3.1 TPM attestation statement certificate requirements
4. 8.4 Android Key Attestation Statement Format
5. 8.5 Android SafetyNet Attestation Statement Format
6. 8.6 FIDO U2F Attestation Statement Format
9. 9 WebAuthn Extensions
1. 9.1 Extension Identifiers
2. 9.2 Defining extensions
3. 9.3 Extending request parameters
4. 9.4 Client extension processing
5. 9.5 Authenticator extension processing
6. 9.6 Example Extension
10. 10 Defined Extensions
1. 10.1 FIDO AppId Extension (appid)
2. 10.2 Simple Transaction Authorization Extension (txAuthSimple)
3. 10.3 Generic Transaction Authorization Extension (txAuthGeneric)
4. 10.4 Authenticator Selection Extension (authnSel)

5. 9.5 Supported Extensions Extension (exts)
6. 9.6 User Verification Index Extension (uvi)
7. 9.7 Location Extension (loc)
8. 9.8 User Verification Method Extension (uvm)

10. 10 IANA Considerations
    1. 10.1 WebAuthn Attestation Statement Format Identifier Registrations
    2. 10.2 WebAuthn Extension Identifier Registrations
    3. 10.3 COSE Algorithm Registrations
11. 11 Sample scenarios
    1. 11.1 Registration
    2. 11.2 Registration Specifically with Platform Authenticator
    3. 11.3 Authentication
    4. 11.4 Decommissioning
12. 12 Acknowledgements
13. Index

    1. Terms defined by this specification
    2. Terms defined by reference
14. References
    1. Normative References
    2. Informative References
15. IDL Index

1. Introduction

This section is not normative.

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

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

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

5. 10.5 Supported Extensions Extension (exts)
6. 10.6 User Verification Index Extension (uvi)
7. 10.7 Location Extension (loc)
8. 10.8 User Verification Method Extension (uvm)

11. 11 IANA Considerations
    1. 11.1 WebAuthn Attestation Statement Format Identifier Registrations
    2. 11.2 WebAuthn Extension Identifier Registrations
    3. 11.3 COSE Algorithm Registrations
12. 12 Sample scenarios
    1. 12.1 Registration
    2. 12.2 Registration Specifically with User Verifying Platform Authenticator
    3. 12.3 Authentication
    4. 12.4 Aborting Authentication Operations
    5. 12.5 Decommissioning
13. 13 Security Considerations
    1. 13.1 Cryptographic Challenges
14. 14 Acknowledgements
15. Index

    1. Terms defined by this specification
    2. Terms defined by reference
16. References
    1. Normative References
    2. Informative References
17. IDL Index
18. Issues Index

1. Introduction

This section is not normative.

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

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

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

1.1. Use Cases

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

1.1. Use Cases

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

1.1.1. Registration

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

1.1.2. Authentication

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

1.1.3. Other use cases and configurations

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

* A user navigates to example.com on their laptop, is guided through a flow to create and register a credential on their phone.
* A user obtains an discrete, roaming authenticator, such as a "fob" with USB or USB+NFC/BLE connectivity options, loads example.com in their browser on a laptop or phone, and is guided though a flow to create and register a credential on the fob.
* A Relying Party prompts the user for their authorization gesture in order to authorize a single transaction, such as a payment or other financial transaction.

2. Conformance

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

2.1. User Agents

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].

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

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

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

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

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

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

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

2.2. Authenticators

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

As described in 1.1 Use Cases, an authenticator may be implemented in the operating system underlying the User Agent, or in external hardware, or a combination of both.

2.3. Relying Parties

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

3. Dependencies

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

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

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

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

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

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

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

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

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

in [HTML52].

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

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

3. Terminology

Assertion
: See Authentication Assertion.

Attestation
: Generally, attestation is a statement serving to bear witness, confirm, or authenticate. In the WebAuthn context, attestation is employed to attest to the provenance of an authenticator and the data it emits; including, for example: credential IDs, credential key pairs, signature counters, etc. An attestation statement is conveyed in an attestation object during registration. See also 5.3 Attestation and Figure 3.

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

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

in [HTML52].

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

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

4. Terminology

Assertion
: See Authentication Assertion.

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

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

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

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

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

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

Authorization Gesture
: An authorization gesture is a physical interaction performed by

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

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

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

Client
: See Conforming User Agent.

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

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

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

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.

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

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

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

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

Credential IDs are generated by authenticators in two forms:

1. At least 16 bytes that include at least 100 bits of entropy,
   or
2. The public key credential source, without its Credential ID, encrypted so only its managing authenticator can decrypt it. This form allows the authenticator to be nearly stateless, by having the Relying Party store any necessary state. Note: [FIDO-UAF-AUTHNR-CMDS] includes guidance on encryption techniques under "Security Guidelines".

Relying Parties do not need to distinguish these two Credential ID forms.

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

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.

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

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

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

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

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

  "Authentication information" (possibly including a private key)
  : Public key credential source

  "Signed value"
  : Authentication assertion

  [RFC4949] "credential"
  : Credential public key or attestation object

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

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

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

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

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

Relying Party Identifier
RP ID
: A valid domain string that identifies the Relying Party on whose behalf a given registration or authentication ceremony is being performed. A public key credential can only be used for authentication with the same entity (as identified by RP ID) it was registered with. By default, the RP ID for a WebAuthn operation is set to the caller's origin's effective domain. This default MAY be overridden by the caller, as long as the caller-specified RP ID value is a registrable domain suffix of or is equal to the caller's origin's effective domain. See also 4.1.3 Create a new credential - PublicKeyCredential's [[Create]](options) method and 4.1.4 Use an existing credential to make an assertion - PublicKeyCredential's [[DiscoverFromExternalSource]](options) method.

  Note: A Public key credential's scope is for a Relying Party's origin, with the following restrictions and relaxations:

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

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

Public Key Credential
: Generically, a credential is data one entity presents to another in order to authenticate the former to the latter [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, who then stores it in conjunction with the present user's account. The authenticator maps the private key portion to the Relying Party's RP ID and stores it.

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

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

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

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

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

  Note: A Public key credential's scope is for a Relying Party's origin, with the following restrictions and relaxations:

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

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

Subsequently, only that Relying Party, as identified by its RP ID, is able to employ the public key credential in authentication ceremonies, via the get() method. The Relying Party uses its stored copy of the credential public key to verify the resultant authentication assertion.

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

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

User Verification
: The technical process by which an authenticator locally authorizes the invocation of the authenticatorMakeCredential and authenticatorGetAssertion operations. User verification may be instigated through various authorization gesture modalities; for example, through a touch plus pin code, password entry, or biometric recognition (e.g., presenting a fingerprint) [ISOBiometricVocabulary]. The intent is to be able to distinguish individual users. Note that invocation of the authenticatorMakeCredential and authenticatorGetAssertion operations implies use of key material managed by the authenticator. Note that for security, user verification and use of credential private keys must occur within a single logical security boundary defining the authenticator.

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

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

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

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

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

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

User Handle
: The user handle is specified by a Relying Party and is a unique identifier for a user account with that Relying Party. A user handle is an opaque byte sequence with a maximum size of 64 bytes.

The user handle is not meant to be displayed to the user, but is used by the Relying Party to control the number of credentials - an authenticator will never contain more than one credential for a given Relying Party under the same user handle.

User Verification
: The technical process by which an authenticator locally authorizes the invocation of the authenticatorMakeCredential and authenticatorGetAssertion operations. User verification may be instigated through various authorization gesture modalities; for example, through a touch plus pin code, password entry, or biometric recognition (e.g., presenting a fingerprint) [ISOBiometricVocabulary]. The intent is to be able to distinguish individual users. Note that invocation of the authenticatorMakeCredential and authenticatorGetAssertion operations implies use of key material managed by the authenticator. Note that for security, user verification and use of credential private keys must occur within a single logical security boundary defining the authenticator.

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

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

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

5. Web Authentication API

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

Party. Scripts can also request the user's permission to perform authentication operations with an existing credential. All such operations are performed in the authenticator and are mediated by the 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 public key credentials belonging to other Relying Parties, each credential is also associated with a Relying Party Identifier, or RP ID. This RP ID is provided by the client to the authenticator for all operations, and the authenticator ensures that credentials created by a Relying Party can only be used in operations requested by the same RP ID. Separating the origin from the RP ID in this way allows the API to be used in cases where a single Relying Party maintains multiple origins.

The client facilitates these security measures by providing the Relying Party's origin and RP ID to the authenticator for each operation. Since this is an integral part of the WebAuthn security model, user agents only expose this API to callers in secure contexts.

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

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 {
    [SameObject] readonly attribute ArrayBuffer              rawId;
    [SameObject] readonly attribute AuthenticatorResponse    response;
    [SameObject] readonly attribute AuthenticationExtensions clientExtensionResu
lts;
};
```

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.

Party. Scripts can also request the user's permission to perform authentication operations with an existing credential. All such operations are performed in the authenticator and are mediated by the browser and/or platform on the user's behalf. At no point does the script get access to the credentials themselves; it only gets information about the credentials in the form of objects.

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

The security properties of this API are provided by the client and the authenticator working together. The authenticator, which holds and manages credentials, ensures that all operations are scoped to a particular origin, and cannot be replayed against a different origin, by incorporating the origin in its responses. Specifically, as defined in 6.2 Authenticator operations, the full origin of the requester is included, and signed over, in the attestation object produced when a new credential is created as well as in all assertions produced by WebAuthn credentials.

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

The client facilitates these security measures by providing the Relying Party's origin and RP ID to the authenticator for each operation. Since this is an integral part of the WebAuthn security model, user agents only expose this API to callers in secure contexts.

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

5.1. PublicKeyCredential Interface

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

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

};
```

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

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

response, of type AuthenticatorResponse, readonly
This attribute contains the authenticator's response to the client's request to either create a public key credential, or 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 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. CredentialCreationOptions Extension

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

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

4.1.2. CredentialRequestOptions Extension

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

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


response, of type AuthenticatorResponse, readonly
This attribute contains the authenticator's response to the client's request to either create a public key credential, or generate an authentication assertion. If the PublicKeyCredential is created in response to create(), this attribute's value will be an AuthenticatorAttestationResponse, otherwise, the PublicKeyCredential was created in response to get(), and this attribute's value will be an AuthenticatorAssertionResponse.

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

[[type]]
The PublicKeyCredential interface object's [[type]] internal slot's value is the string "public-key".

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

[[discovery]]
The PublicKeyCredential interface object's [[discovery]] internal slot's value is "remote".

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

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

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

5.1.1. CredentialCreationOptions Extension

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

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

5.1.2. CredentialRequestOptions Extension

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

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

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 object.

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

This method accepts a single argument:

options
: This argument is a CredentialCreationOptions object whose options.publicKey member contains a MakePublicKeyCredentialOptions object specifying the desired attributes of the to-be-created public key credential.

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's 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.

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

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

This internal method accepts three arguments:

origin
: This argument is the relevant settings object's origin, as determined by the calling create() implementation.

options
: This argument is a CredentialCreationOptions object whose options.publicKey member contains a MakePublicKeyCredentialOptions object specifying the desired attributes of the to-be-created public key credential.

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

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

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

1. Assert: options.publicKey is present.
2. If sameOriginWithAncestors is false, return a "NotAllowedError" DOMException.
   Note: This "sameOriginWithAncestors" restriction aims to address the concern raised in the Origin Confusion section of [CREDENTIAL-MANAGEMENT-1], while allowing Relying Party script access to Web Authentication functionality, e.g., when running in a secure context framed document that is same-origin with its ancestors. However, in the future, this specification (in conjunction with [CREDENTIAL-MANAGEMENT-1]) may provide Relying Parties with more fine-grained control--e.g., ranging from allowing only top-level access to Web Authentication functionality, to allowing cross-origin embedded cases--by leveraging [Feature-Policy] once the latter specification becomes stably implemented in user agents.
3. Let options be the value of options.publicKey.
4. If the timeout member of options is present, check if its value lies within a reasonable range as defined by the platform and if not, correct it to the closest value lying within that range. Set a timer lifetimeTimer to this adjusted value. If the timeout member of options is not present, then set lifetimeTimer to a platform-specific default.
5. Let callerOrigin be origin. If callerOrigin is an opaque origin, return a DOMException whose name is "NotAllowedError", and terminate this algorithm.
6. Let effectiveDomain be the callerOrigin's effective domain. If

7. Let effectiveDomain be the callerOrigin's effective domain. If effective domain is not a valid domain, then return a DOMException whose name is "SecurityError" and terminate this algorithm.
Note: An effective domain may resolve to a host, which can be represented in various manners, such as domain, ipv4 address, ipv6 address, opaque host, or empty host. Only the domain format of host is allowed here.
8. Let rpId be effectiveDomain.
9. If options.rp.id is present:
1. 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.
2. Set rpId to options.rp.id.
Note: rpId represents the caller's RP ID. The RP ID defaults to being the caller's origin's effective domain unless the caller has explicitly set options.rp.id when calling create().
10. Let credTypesAndPubKeyAlgs be a new list whose items are pairs of PublicKeyCredentialType and a COSEAlgorithmIdentifier.
11. For each current of options.pubKeyCredParams:
1. If current.type does not contain a PublicKeyCredentialType supported by this implementation, then continue.
2. Let alg be current.alg.
3. Append the pair of current.type and alg to credTypesAndPubKeyAlgs.
12. If credTypesAndPubKeyAlgs is empty and options.pubKeyCredParams is not empty, cancel the timer started in step 2, return a DOMException whose name is "NotSupportedError", and terminate this algorithm.
13. Let clientExtensions be a new map and let authenticatorExtensions be a new map.
14. 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.
15. Let collectedClientData be a new CollectedClientData instance whose fields are:

challenge
The base64url encoding of options.challenge.

origin
The serialization of callerOrigin.

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

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

clientExtensions

effective domain is not a valid domain, then return a DOMException whose name is "SecurityError" and terminate this algorithm.
Note: An effective domain may resolve to a host, which can be represented in various manners, such as domain, ipv4 address, ipv6 address, opaque host, or empty host. Only the domain format of host is allowed here.
7. If options.rp.id

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

Is not present
Set options.rp.id to effectiveDomain.

Note: options.rp.id represents the caller's RP ID. The RP ID defaults to being the caller's origin's effective domain unless the caller has explicitly set options.rp.id when calling create().
8. Let credTypesAndPubKeyAlgs be a new list whose items are pairs of PublicKeyCredentialType and a COSEAlgorithmIdentifier.
9. For each current of options.pubKeyCredParams:
1. If current.type does not contain a PublicKeyCredentialType supported by this implementation, then continue.
2. Let alg be current.alg.
3. Append the pair of current.type and alg to credTypesAndPubKeyAlgs.
10. If credTypesAndPubKeyAlgs is empty and options.pubKeyCredParams is not empty, return a DOMException whose name is "NotSupportedError", and terminate this algorithm.
11. Let clientExtensions be a new map and let authenticatorExtensions be a new map.
12. If the extensions member of options is present, then for each extensionId -> clientExtensionInput of options.extensions:
1. If extensionId is not supported by this client platform or is not a registration extension, then continue.
2. Set clientExtensions[extensionId] to clientExtensionInput.
3. If extensionId is not an authenticator extension, then continue.
4. Let authenticatorExtensionInput be the (CBOR) result of running extensionId's client extension processing algorithm on clientExtensionInput. If the algorithm returned an error, continue.
5. Set authenticatorExtensions[extensionId] to the base64url encoding of authenticatorExtensionInput.
13. Let collectedClientData be a new CollectedClientData instance whose fields are:

type
The string "webauthn.create".

challenge
The base64url encoding of options.challenge.

origin
The serialization of callerOrigin.

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

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

clientExtensions

clientExtensions

authenticatorExtensions
authenticatorExtensions

16. Let clientDataJSON be the JSON-serialized client data constructed from collectedClientData.
17. Let clientDataHash be the hash of the serialized client data represented by clientDataJSON.
18. Let currentlyAvailableAuthenticators be a new ordered set consisting of all authenticators currently available on this platform.
19. Let selectedAuthenticators be a new ordered set.
20. If currentlyAvailableAuthenticators is empty, return a DOMException whose name is "NotFoundError", and terminate this algorithm.
21. If options.authenticatorSelection is present, iterate through currentlyAvailableAuthenticators and do the following for each authenticator:
1. If aa is present and its value is not equal to authenticator's attachment modality, continue.
2. If rk is set to true and the authenticator is not capable of storing a Client-Side-Resident Credential Private Key, continue.
3. If uv is set to true and the authenticator is not capable of performing user verification, continue.
4. Append authenticator to selectedAuthenticators.
22. If selectedAuthenticators is empty, return a DOMException whose name is "ConstraintError", and terminate this algoritm.
23. Let issuedRequests be a new ordered set.
24. For each authenticator in currentlyAvailableAuthenticators:
1. Let excludeCredentialDescriptorList be a new list.
2. For each credential descriptor C in options.excludeCredentials:
1. If C.transports is not empty, and authenticator is connected over a transport not mentioned in C.transports, the client MAY continue.
2. Otherwise, Append C to excludeCredentialDescriptorList.
3. In parallel, invoke the authenticatorMakeCredential operation on authenticator with rpId, clientDataHash, options.rp, options.user, options.authenticatorSelection.rk, credTypesAndPubKeyAlgs, excludeCredentialDescriptorList, and authenticatorExtensions as parameters.
4. Append authenticator to issuedRequests.
25. 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.
26. While issuedRequests is not empty, perform the following actions

clientExtensions

authenticatorExtensions
authenticatorExtensions

14. Let clientDataJSON be the JSON-serialized client data constructed from collectedClientData.
15. Let clientDataHash be the hash of the serialized client data represented by clientDataJSON.
16. If the options.signal is present and its aborted flag is set to true, return a DOMException whose name is "AbortError" and terminate this algorithm.
17. Start lifetimeTimer.
18. Let issuedRequests be a new ordered set.
19. For each authenticator that becomes available on this platform during the lifetime of lifetimeTimer, do the following:
The definitions of "lifetime of" and "becomes available" are intended to represent how devices are hotplugged into (USB) or discovered by (NFC) browsers, and are under-specified. Resolving this with good definitions or some other means will be addressed by resolving Issue #613.
1. If options.authenticatorSelection is present:
1. If options.authenticatorSelection.authenticatorAttachment is present and its value is not equal to authenticator's attachment modality, continue.
2. If options.authenticatorSelection.requireResidentKey is set to true and the authenticator is not capable of storing a Client-Side-Resident Credential Private Key, continue.
3. If options.authenticatorSelection.userVerification is set to required and the authenticator is not capable of performing user verification, continue.
2. Let userVerification be the effective user verification requirement for credential creation, a Boolean value, as follows. If options.authenticatorSelection.userVerification
is set to required
Let userVerification be true.
is set to preferred
If the authenticator
is capable of user verification
Let userVerification be true.
is not capable of user verification
Let userVerification be false.
is set to discouraged
Let userVerification be false.
3. Let userPresence be a Boolean value set to the inverse of userVerification.
4. Let excludeCredentialDescriptorList be a new list.
5. For each credential descriptor C in options.excludeCredentials:
1. If C.transports is not empty, and authenticator is connected over a transport not mentioned in C.transports, the client MAY continue.
2. Otherwise, Append C to excludeCredentialDescriptorList.
6. Invoke the authenticatorMakeCredential operation on authenticator with clientDataHash, options.rp, options.user, options.authenticatorSelection.requireResidentKey, userPresence, userVerification, credTypesAndPubKeyAlgs, excludeCredentialDescriptorList, and authenticatorExtensions as parameters.
7. Append authenticator to issuedRequests.
20. While issuedRequests is not empty, perform the following actions depending upon lifetimeTimer and responses from the authenticators:

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:

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

If the options.signal is present and its aborted flag is set to true,
For each authenticator in issuedRequests invoke the authenticatorCancel operation on authenticator and remove authenticator from issuedRequests. Then return a DOMException whose name is "AbortError" and terminate this algorithm.

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

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

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

If any authenticator indicates success,

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

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

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

clientDataJSONResult
whose value is the bytes of clientDataJSON.

attestationConveyancePreferenceOption
whose value is the value of options.attestation.

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

3. Let constructCredentialAlg be an algorithm that takes a global object global, and whose steps are:
   1. Let attestationObject be a new ArrayBuffer, created using global's %ArrayBuffer%, containing the bytes of credentialCreationData.attestationObjectResult's value.
   2. If credentialCreationData.attestationConveyancePreferenceOption's value is

"none"
Replace potentially uniquely identifying information (such as AAGUID and attestation certificates) in the

[[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.

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

attested credential data and attestation statement, respectively, with blinded versions of the same data.

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

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

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

@balfanz wishes to add to the "direct" case: If the authenticator violates the privacy requirements of the attestation type it is using, the client SHOULD terminate this algorithm with a "AttestationNotPrivateError".

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

[[identifier]]
id

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

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

attestationObject
attestationObject

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

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

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

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

4.1.4. Use an existing credential to make an assertion - PublicKeyCredential's [[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 handled by navigator.credentials.get().

This method accepts a single argument:

options
: This argument is a CredentialRequestOptions object whose options.publicKey member contains a challenge and additional options as described in 4.5 Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions). The selected authenticator signs the challenge along with other collected data in order to produce an assertion. See 5.2.2 The authenticatorGetAssertion operation.

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 the timeout member of options is present, check if its value

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

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

Relying Parties call navigator.credentials.get({publicKey:..., ...}) to discover and use an existing public key credential, with the user's consent. Relying Party script optionally specifies some criteria to indicate what credential sources are acceptable to it. The user agent and/or platform locates credential sources matching the specified criteria, and guides the user to pick one that the script will be allowed to use. The user may choose to decline the entire interaction even if a credential source is present, for example to maintain privacy. If the user picks a credential source, the user agent then uses 6.2.2 The authenticatorGetAssertion operation to sign a Relying Party-provided challenge and other collected data into an assertion, which is used as a credential.

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

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

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

This internal method accepts three arguments:

origin
: This argument is the relevant settings object's origin, as determined by the calling get() implementation, i.e., CredentialsContainer's Request a Credential abstract operation.

options
: This argument is a CredentialRequestOptions object whose options.publicKey member contains a PublicKeyCredentialRequestOptions object specifying the desired attributes of the public key credential to discover.

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

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

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

1. Assert: options.publicKey is present.
2. If sameOriginWithAncestors is false, return a "NotAllowedError" DOMException.
   Note: This "sameOriginWithAncestors" restriction aims to address the concern raised in the Origin Confusion section of [CREDENTIAL-MANAGEMENT-1], while allowing Relying Party script access to Web Authentication functionality, e.g., when running in a secure context framed document that is same-origin with its ancestors. However, in the future, this specification (in

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.

4. Let global be the PublicKeyCredential's interface object's environment settings object's global object.
5. 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.
6. Let effectiveDomain be the callerOrigin's effective domain. If effective domain is not a valid domain, then return a DOMException whose name is "SecurityError" and terminate this algorithm. Note: An effective domain may resolve to a host, which can be represented in various manners, such as domain, ipv4 address, ipv6 address, opaque host, or empty host. Only the domain format of host is allowed here.
7. If options.rpId is not present, then set rpId to effectiveDomain. Otherwise:
    1. If options.rpId is not a registrable domain suffix of and is not equal to effectiveDomain, return a DOMException whose name is "SecurityError", and terminate this algorithm.
    2. Set rpId to options.rpId. Note: rpId represents the caller's RP ID. The RP ID defaults to being the caller's origin's effective domain unless the caller has explicitly set options.rpId when calling get().
8. Let clientExtensions be a new map and let authenticatorExtensions be a new map.
9. If the extensions member of options is present, then for each extensionId -> clientExtensionInput of options.extensions:
    1. If extensionId is not supported by this client platform or is not an authentication extension, then continue.
    2. Set clientExtensions[extensionId] to clientExtensionInput.
    3. If extensionId is not an authenticator extension, then continue.
    4. Let authenticatorExtensionInput be the (CBOR) result of running extensionId's client extension processing algorithm on clientExtensionInput. If the algorithm returned an error, continue.
    5. Set authenticatorExtensions[extensionId] to the base64url encoding of authenticatorExtensionInput.
10. Let collectedClientData be a new CollectedClientData instance whose fields are:

    challenge
        The base64url encoding of options.challenge

    origin
        The serialization of callerOrigin.

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

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

conjunction with [CREDENTIAL-MANAGEMENT-1]) may provide Relying Parties with more fine-grained control--e.g., ranging from allowing only top-level access to Web Authentication functionality, to allowing cross-origin embedded cases--by leveraging [Feature-Policy] once the latter specification becomes stably implemented in user agents.

3. Let options be the value of options.publicKey.
4. If the timeout member of options is present, check if its value lies within a reasonable range as defined by the platform and if not, correct it to the closest value lying within that range. Set a timer lifetimeTimer to this adjusted value. If the timeout member of options is not present, then set lifetimeTimer to a platform-specific default.
5. Let callerOrigin be origin. If callerOrigin is an opaque origin, return a DOMException whose name is "NotAllowedError", and terminate this algorithm.
6. Let effectiveDomain be the callerOrigin's effective domain. If effective domain is not a valid domain, then return a DOMException whose name is "SecurityError" and terminate this algorithm. Note: An effective domain may resolve to a host, which can be represented in various manners, such as domain, ipv4 address, ipv6 address, opaque host, or empty host. Only the domain format of host is allowed here.
7. If options.rpId is not present, then set rpId to effectiveDomain. Otherwise:
    1. If options.rpId is not a registrable domain suffix of and is not equal to effectiveDomain, return a DOMException whose name is "SecurityError", and terminate this algorithm.
    2. Set rpId to options.rpId. Note: rpId represents the caller's RP ID. The RP ID defaults to being the caller's origin's effective domain unless the caller has explicitly set options.rpId when calling get().
8. Let clientExtensions be a new map and let authenticatorExtensions be a new map.
9. If the extensions member of options is present, then for each extensionId -> clientExtensionInput of options.extensions:
    1. If extensionId is not supported by this client platform or is not an authentication extension, then continue.
    2. Set clientExtensions[extensionId] to clientExtensionInput.
    3. If extensionId is not an authenticator extension, then continue.
    4. Let authenticatorExtensionInput be the (CBOR) result of running extensionId's client extension processing algorithm on clientExtensionInput. If the algorithm returned an error, continue.
    5. Set authenticatorExtensions[extensionId] to the base64url encoding of authenticatorExtensionInput.
10. Let collectedClientData be a new CollectedClientData instance whose fields are:

    type
        The string "webauthn.get".

    challenge
        The base64url encoding of options.challenge

    origin
        The serialization of callerOrigin.

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

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

clientExtensions
clientExtensions

authenticatorExtensions
authenticatorExtensions

11. Let clientDataJSON be the JSON-serialized client data constructed from collectedClientData.
12. Let clientDataHash be the hash of the serialized client data represented by clientDataJSON.
13. Let issuedRequests be a new ordered set.
14. If there are no authenticators currently available on this platform, return a DOMException whose name is "NotFoundError", and terminate this algorithm.

15. Let authenticator be a platform-specific handle whose value identifies an authenticator.
16. For each authenticator currently available on this platform, perform the following steps:
1. Let allowCredentialDescriptorList be a new list.
2. If options.allowCredentials is not empty, execute a

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

is not empty

1. Let distinctTransports be a new ordered set.
2. For each credential descriptor C in

clientExtensions
clientExtensions

authenticatorExtensions
authenticatorExtensions

11. Let clientDataJSON be the JSON-serialized client data constructed from collectedClientData.
12. Let clientDataHash be the hash of the serialized client data represented by clientDataJSON.
13. If the options.signal is present and its aborted flag is set to true, return a DOMException whose name is "AbortError" and

terminate this algorithm.
14. Let issuedRequests be a new ordered set.
15. Let authenticator be a platform-specific handle whose value identifies an authenticator.
16. Start lifetimeTimer.
17. For each authenticator that becomes available on this platform during the lifetime of lifetimeTimer, perform the following steps: The definitions of "lifetime of" and "becomes available" are intended to represent how devices are hotplugged into (USB) or discovered by (NFC) browsers, and are under-specified. Resolving this with good definitions or some other means will be addressed by resolving Issue #613.
1. If options.userVerification is set to required and the authenticator is not capable of performing user verification, continue.
2. Let userVerification be the effective user verification requirement for assertion, a Boolean value, as follows. If options.userVerification

is set to required
Let userVerification be true.

is set to preferred
If the authenticator

is capable of user verification
Let userVerification be true.

is not capable of user verification
Let userVerification be false.

is set to discouraged
Let userVerification be false.

3. Let userPresence be a Boolean value set to the inverse of userVerification.
4. Let allowCredentialDescriptorList be a new list.
5. If options.allowCredentials is not empty, execute a platform-specific procedure to determine which, if any, public key credentials described by options.allowCredentials are bound to this authenticator, by matching with rpId, options.allowCredentials.id, and options.allowCredentials.type. Set allowCredentialDescriptorList to this filtered list.
6. If allowCredentialDescriptorList

is not empty

1. Let distinctTransports be a new ordered set.
2. If allowCredentialDescriptorList has exactly one value, let savedCredentialId be a new PublicKeyCredentialDescriptor.id and set its value to allowCredentialDescriptorList[0].id's value (see here in 6.2.2 The authenticatorGetAssertion operation for more information).

allowCredentialDescriptorList, append each value, if any, of C.transports to distinctTransports.
Note: This will aggregate only distinct values of transports (for this authenticator) in distinctTransports due to the properties of ordered sets.

3. If distinctTransports

is not empty
The client selects one transport value from distinctTransports, possibly incorporating local configuration knowledge of the appropriate transport to use with authenticator in making its selection.

Then, using transport, invoke in parallel the authenticatorGetAssertion operation on authenticator, with rpId, clientDataHash, allowCredentialDescriptorList, and

authenticatorExtensions as parameters.

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

clientExtensions as parameters.

is empty
Using local configuration knowledge of the appropriate transport to use with authenticator, invoke in parallel the authenticatorGetAssertion operation on authenticator with rpId, clientDataHash, and clientExtensions as parameters.

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

4. Append authenticator to issuedRequests.

17. 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.

18. 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.

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

1. For each credential descriptor C in
allowCredentialDescriptorList, append each value, if any, of C.transports to distinctTransports.
Note: This will aggregate only distinct values of transports (for this authenticator) in distinctTransports due to the properties of ordered sets.

2. If distinctTransports

is not empty
The client selects one transport value from distinctTransports, possibly incorporating local configuration knowledge of the appropriate transport to use with authenticator in making its selection.

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

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

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

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

7. Append authenticator to issuedRequests.

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

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

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

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 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 5.2.2 The authenticatorGetAssertion operation.

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

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

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

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

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

If any authenticator indicates success,

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

credentialIdResult
If savedCredentialId exists, set the value of credentialIdResult to be the bytes of savedCredentialId. Otherwise, set the value of credentialIdResult to be the bytes of the credential ID returned from the successful authenticatorGetAssertion operation, as defined in 6.2.2 The authenticatorGetAssertion operation.

clientDataJSONResult
whose value is the bytes of clientDataJSON.

authenticatorDataResult
whose value is the bytes of the authenticator data returned by the authenticator.

signatureResult
whose value is the bytes of the signature value returned by the authenticator.

userHandleResult
whose value is the bytes of the user handle returned by the authenticator.

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

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

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

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.

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

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

4.1.5. Platform Authenticator Availability - PublicKeyCredential's isPlatformAuthenticatorAvailable() method

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

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

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

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

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

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

2. Return pubKeyCred.
4. For each remaining authenticator in issuedRequests invoke the authenticatorCancel operation on authenticator and remove it from issuedRequests.
5. Return constructAssertionAlg and terminate this algorithm.

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

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

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

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

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

This internal method accepts two arguments:

credential
This argument is a PublicKeyCredential object.

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

* Whether the user is running in private or incognito mode.
* Whether the user has configured the client to not create such credentials.
* Whether the user has previously expressed an unwillingness to create a new credential for this Relying Party, either through configuration or by declining a user interface prompt.
* The user's explicitly stated intentions, determined through user interaction.

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

This method has no arguments and returns a boolean value.

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

```
[SecureContext]
partial interface PublicKeyCredential {
    [Unscopable] Promise < boolean > isPlatformAuthenticatorAvailable();
};
```

4.2. Authenticator Responses (interface AuthenticatorResponse)

Authenticators respond to Relying Party requests by returning an object derived from the AuthenticatorResponse interface:

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

clientDataJSON, of type ArrayBuffer, readonly
: This attribute contains a JSON serialization of the client data passed to the authenticator by the client in its call to either create() or get().

4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

The AuthenticatorAttestationResponse interface represents the

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

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

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

5.1.6. Availability of User-Verifying Platform Authenticator - PublicKeyCredential's isUserVerifyingPlatformAuthenticatorAvailable() method

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

* Whether the user is running in private or incognito mode.
* Whether the user has configured the client to not create such credentials.
* Whether the user has previously expressed an unwillingness to create a new credential for this Relying Party, either through configuration or by declining a user interface prompt.
* The user's explicitly stated intentions, determined through user interaction.

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

This method has no arguments and returns a boolean value.

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

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

5.2. Authenticator Responses (interface AuthenticatorResponse)

Authenticators respond to Relying Party requests by returning an object derived from the AuthenticatorResponse interface:

```
[SecureContext, Exposed=Window]
interface AuthenticatorResponse {
    [SameObject] readonly attribute ArrayBuffer      clientDataJSON;
};
```

clientDataJSON, of type ArrayBuffer, readonly
: This attribute contains a JSON serialization of the client data passed to the authenticator by the client in its call to either create() or get().

5.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

The AuthenticatorAttestationResponse interface represents the

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

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

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

attestationObject, of type ArrayBuffer, readonly
: This attribute contains an attestation object, which is opaque to, and cryptographically protected against tampering by, the client. The attestation object contains both authenticator data and an attestation statement. The former contains the AAGUID, a unique credential ID, and the credential public key. The contents of the attestation statement are determined by the attestation statement format used by the authenticator. It also contains any additional information that the Relying Party's server requires to validate the attestation statement, as well as to decode and validate the authenticator data along with the JSON-serialized client data. For more details, see 5.3 Attestation, 5.3.4 Generating an Attestation Object, and 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 {
    [SameObject] readonly attribute ArrayBuffer      authenticatorData;
    [SameObject] readonly attribute ArrayBuffer      signature;

};
```

clientDataJSON
: This attribute, inherited from AuthenticatorResponse, contains the JSON-serialized client data (see 4.7.1 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

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

```
[SecureContext, Exposed=Window]
interface AuthenticatorAttestationResponse : AuthenticatorResponse {
    [SameObject] readonly attribute ArrayBuffer      attestationObject;
};
```

clientDataJSON
: This attribute, inherited from AuthenticatorResponse, contains the JSON-serialized client data (see 6.3 Attestation) passed to the authenticator by the client in order to generate this credential. The exact JSON serialization must be preserved, as the hash of the serialized client data has been computed over it.

attestationObject, of type ArrayBuffer, readonly
: This attribute contains an attestation object, which is opaque to, and cryptographically protected against tampering by, the client. The attestation object contains both authenticator data and an attestation statement. The former contains the AAGUID, a unique credential ID, and the credential public key. The contents of the attestation statement are determined by the attestation statement format used by the authenticator. It also contains any additional information that the Relying Party's server requires to validate the attestation statement, as well as to decode and validate the authenticator data along with the JSON-serialized client data. For more details, see 6.3 Attestation, 6.3.4 Generating an Attestation Object, and Figure 3.

5.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)

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

```
[SecureContext, Exposed=Window]
interface AuthenticatorAssertionResponse : AuthenticatorResponse {
    [SameObject] readonly attribute ArrayBuffer      authenticatorData;
    [SameObject] readonly attribute ArrayBuffer      signature;
    [SameObject] readonly attribute ArrayBuffer      userHandle;
};
```

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

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

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

userHandle, of type ArrayBuffer, readonly
: This attribute contains the user handle returned from the authenticator. See 6.2.2 The authenticatorGetAssertion

PublicKeyCredentialParameters)

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

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

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

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

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

4.4. Options for Credential Creation (dictionary MakePublicKeyCredentialOptions)

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

    required BufferSource                      challenge;
    required sequence<PublicKeyCredentialParameters>  pubKeyCredParams;

    unsigned long                              timeout;
    sequence<PublicKeyCredentialDescriptor>    excludeCredentials = [];
    AuthenticatorSelectionCriteria             authenticatorSelection;

    AuthenticationExtensions                   extensions;
};
```

rp, of type PublicKeyCredentialEntity
: This member contains data about the Relying Party responsible for the request.

  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 omitted, its value will be the CredentialsContainer object's relevant settings object's origin's effective domain.

user, of type PublicKeyCredentialUserEntity
: This member contains data about the user account for which the Relying Party is requesting attestation.

  Its value's name member 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

operation.

5.3. Parameters for Credential Generation (dictionary PublicKeyCredentialParameters)

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

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

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

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

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

5.4. Options for Credential Creation (dictionary MakePublicKeyCredentialOptions)

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

    required BufferSource                      challenge;
    required sequence<PublicKeyCredentialParameters>  pubKeyCredParams;

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

rp, of type PublicKeyCredentialRpEntity
: This member contains data about the Relying Party responsible for the request.

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

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

user, of type PublicKeyCredentialUserEntity
: This member contains data about the user account for which the Relying Party is requesting attestation.

  Its value's name member contains a name for the user account (e.g., "john.p.smith@example.com" or "+14255551234").

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

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

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.

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

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

excludeCredentials, of type sequence<PublicKeyCredentialDescriptor>, defaulting to None
This member is intended for use by Relying Parties that wish to limit the creation of multiple credentials for the same account on a single authenticator. The platform is requested to return an error if the new credential would be created on an authenticator that also contains one of the credentials enumerated in this parameter.

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

4.4.1. Public Key Entity Description (dictionary PublicKeyCredentialEntity)

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

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

id, of type DOMString
A unique identifier for the entity. For a relying party entity, sets the RP ID. For a user account entity, this will be an arbitrary string specified by the relying party.

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.

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

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

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

excludeCredentials, of type sequence<PublicKeyCredentialDescriptor>, defaulting to None
This member is intended for use by Relying Parties that wish to limit the creation of multiple credentials for the same account on a single authenticator. The platform is requested to return an error if the new credential would be created on an authenticator that also contains one of the credentials enumerated in this parameter.

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

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

extensions, of type AuthenticationExtensions
This member contains additional parameters requesting additional processing by the client and authenticator. For example, the caller may request that only authenticators with certain capabilies be used to create the credential, or that particular information be returned in the attestation object. Some extensions are defined in 9 WebAuthn Extensions; consult the IANA "WebAuthn Extension Identifier" registry established by [WebAuthn-Registries] for an up-to-date list of registered WebAuthn Extensions.

5.4.1. Public Key Entity Description (dictionary PublicKeyCredentialEntity)

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

```
dictionary PublicKeyCredentialEntity {
    required DOMString    name;

    USVString             icon;
};
```

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

```
1472
1473        icon, of type USVString
1474            A serialized URL which resolves to an image associated with the
1475            entity. For example, this could be a user's avatar or a Relying
1476            Party's logo.
1477
1478        4.4.2. User Account Parameters for Credential Generation (dictionary
1479        PublicKeyCredentialUserEntity)
1480
1481        The PublicKeyCredentialUserEntity dictionary is used to supply
1482        additional user account attributes when creating a new credential.
1483    dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
1484        DOMString     displayName;
1485    };
1486
1487        displayName, of type DOMString
1488            A friendly name for the user account (e.g., "John P. Smith").
1489
1490        4.4.3. Authenticator Selection Criteria (dictionary
1491        AuthenticatorSelectionCriteria)
1492
1493        Relying Parties may use the AuthenticatorSelectionCriteria dictionary
1494        to specify their requirements regarding authenticator attributes.
1495    dictionary AuthenticatorSelectionCriteria {
1496        AuthenticatorAttachment     aa;       // authenticatorAttachment
1497        boolean                     rk = false; // requireResidentKey
1498        boolean                     uv = false; // requireUserVerification
1499    };
1500
1501        aa (authenticatorAttachment), of type AuthenticatorAttachment
1502            If this member is present, eligible authenticators are filtered
1503            to only authenticators attached with the specified 4.4.4
1504            Authenticator Attachment enumeration (enum
1505            AuthenticatorAttachment).
1506
1507        rk (requireResidentKey), of type boolean, defaulting to false
1508            This member describes the Relying Parties' requirements
1509            regarding availability of the Client-side-resident Credential
1510            Private Key. If the parameter is set to true, the authenticator
1511            MUST create a Client-side-resident Credential Private Key when
1512            creating a public key credential.
1513
1514        uv (requireUserVerification), of type boolean, defaulting to false
1515            This member describes the Relying Parties' requirements
1516            regarding the authenticator being capable of performing user
1517            verification. If the parameter is set to true, the authenticator
```

```
1891
1892        icon, of type USVString
1893            A serialized URL which resolves to an image associated with the
1894            entity. For example, this could be a user's avatar or a Relying
1895            Party's logo. This URL MUST be an a priori authenticated URL.
1896            Authenticators MUST accept and store a 128 byte minimum length
1897            for a icon members's value. Authenticators MAY ignore a icon
1898            members's value if its length is greater than 128 byes.
1899
1900        5.4.2. RP Parameters for Credential Generation (dictionary
1901        PublicKeyCredentialRpEntity)
1902
1903        The PublicKeyCredentialRpEntity dictionary is used to supply additional
1904        Relying Party attributes when creating a new credential.
1905    dictionary PublicKeyCredentialRpEntity : PublicKeyCredentialEntity {
1906        DOMString     id;
1907    };
1908
1909        id, of type DOMString
1910            A unique identifier for the Relying Party entity, which sets the
1911            RP ID.
1912
1913        5.4.3. User Account Parameters for Credential Generation (dictionary
1914        PublicKeyCredentialUserEntity)
1915
1916        The PublicKeyCredentialUserEntity dictionary is used to supply
1917        additional user account attributes when creating a new credential.
1918    dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
1919        required BufferSource   id;
1920        required DOMString      displayName;
1921    };
1922
1923        id, of type BufferSource
1924            The user handle of the user account entity.
1925
1926        displayName, of type DOMString
1927            A friendly name for the user account (e.g., "John P. Smith").
1928            Authenticators MUST accept and store a 64 byte minimum length
1929            for a displayName members's value. Authenticators MAY truncate a
1930            displayName member's value to a length equal to or greater than
1931            64 bytes.
1932
1933        5.4.4. Authenticator Selection Criteria (dictionary
1934        AuthenticatorSelectionCriteria)
1935
1936        Relying Parties may use the AuthenticatorSelectionCriteria dictionary
1937        to specify their requirements regarding authenticator attributes.
1938    dictionary AuthenticatorSelectionCriteria {
1939        AuthenticatorAttachment     authenticatorAttachment;
1940        boolean                     requireResidentKey = false;
1941        UserVerificationRequirement  userVerification = "preferred";
1942    };
1943
1944        authenticatorAttachment, of type AuthenticatorAttachment
1945            If this member is present, eligible authenticators are filtered
1946            to only authenticators attached with the specified 5.4.5
1947            Authenticator Attachment enumeration (enum
1948            AuthenticatorAttachment).
1949
1950        requireResidentKey, of type boolean, defaulting to false
1951            This member describes the Relying Parties' requirements
1952            regarding availability of the Client-side-resident Credential
1953            Private Key. If the parameter is set to true, the authenticator
1954            MUST create a Client-side-resident Credential Private Key when
1955            creating a public key credential.
1956
1957        userVerification, of type UserVerificationRequirement, defaulting to
1958            "preferred"
1959            This member describes the Relying Party's requirements regarding
1960            user verification for the create() operation. Eligible
```

MUST perform user verification when performing the create() operation and future 4.1.4 Use an existing credential to make an assertion - PublicKeyCredential's [[DiscoverFromExternalSource]](options) method operations when it is requested to verify the credential.

Note: These identifiers are intentionally short, rather than descriptive, because they will be serialized into a message to the authenticator, which may be sent over a low-bandwidth link.

4.4.4. Authenticator Attachment enumeration (enum AuthenticatorAttachment)

```
enum AuthenticatorAttachment {
    "plat",  // Platform attachment
    "xplat"  // Cross-platform attachment
};
```

Clients may communicate with authenticators using a variety of mechanisms. For example, a client may use a platform-specific API to communicate with an authenticator which is physically bound to a platform. On the other hand, a client may use a variety of standardized cross-platform transport protocols such as Bluetooth (see 4.7.4 Authenticator Transport enumeration (enum AuthenticatorTransport)) to discover and communicate with cross-platform attached authenticators. Therefore, we use AuthenticatorAttachment to describe an authenticator's attachment modality. We define authenticators that are part of the client's platform as having a platform attachment, and refer to them as platform authenticators. While those that are reachable via cross-platform transport protocols are defined as having cross-platform attachment, and refer to them as roaming authenticators.

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

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

4.5. Options for Assertion Generation (dictionary

authenticators are filtered to only those capable of satisfying this requirement.

5.4.5. Authenticator Attachment enumeration (enum AuthenticatorAttachment)

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

Clients may communicate with authenticators using a variety of mechanisms. For example, a client may use a platform-specific API to communicate with an authenticator which is physically bound to a platform. On the other hand, a client may use a variety of standardized cross-platform transport protocols such as Bluetooth (see 5.8.4 Authenticator Transport enumeration (enum AuthenticatorTransport)) to discover and communicate with cross-platform attached authenticators. Therefore, we use AuthenticatorAttachment to describe an authenticator's attachment modality. We define authenticators that are part of the client's platform as having a platform attachment, and refer to them as platform authenticators. While those that are reachable via cross-platform transport protocols are defined as having cross-platform attachment, and refer to them as roaming authenticators.

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

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

5.4.6. Attestation Conveyance Preference enumeration (enum AttestationConveyancePreference)

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

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

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

PublicKeyCredentialRequestOptions)

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

```
dictionary PublicKeyCredentialRequestOptions {
    required BufferSource               challenge;
    unsigned long                       timeout;
    USVString                           rpId;
    sequence<PublicKeyCredentialDescriptor> allowCredentials = [];

    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.

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

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

allowCredentials, of type sequence<PublicKeyCredentialDescriptor>, defaulting to None
: This optional member contains a list of PublicKeyCredentialDescriptor 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 list).

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.

4.6. Authentication Extensions (typedef AuthenticationExtensions)

conveyance yielding verifiable attestation statements, but allows the client to decide how to obtain such attestation statements. The client may replace the authenticator-generated attestation statements with attestation statements generated by a Privacy CA, in order to protect the user's privacy, or to assist Relying Parties with attestation verification in a heterogeneous ecosystem. Note: There is no guarantee that the Relying Party will obtain a verifiable attestation statement in this case. For example, in the case that the authenticator employs self attestation.

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

5.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)

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

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

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

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

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

allowCredentials, of type sequence<PublicKeyCredentialDescriptor>, defaulting to None
: This optional member contains a list of PublicKeyCredentialDescriptor objects representing public key credentials acceptable to the caller, in decending order of the caller's preference (the first item in the list is the most preferred credential, and so on down the list).

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

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

5.6. Abort operations with AbortSignal

Developers are encouraged to leverage the AbortController to manage the [[Create]](origin, options, sameOriginWithAncestors) and [[DiscoverFromExternalSource]](origin, options,

```
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.

4.7. Supporting Data Structures

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

4.7.1. 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          hashAlgorithm;
    DOMString                   tokenBindingId;
    AuthenticationExtensions    clientExtensions;
    AuthenticationExtensions    authenticatorExtensions;
};
```

sameOriginWithAncestors) operations. See DOM 3.3 Using AbortController and AbortSignal objects in APIs section for detailed instructions.

Note: DOM 3.3 Using AbortController and AbortSignal objects in APIs section specifies that web platform APIs integrating with the AbortController must reject the promise immediately once the aborted flag is set. Given the complex inheritance and parallelization structure of the [[Create]](origin, options, sameOriginWithAncestors) and [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) methods, the algorithms for the two APIs fulfills this requirement by checking the aborted flag in three places. In the case of [[Create]](origin, options, sameOriginWithAncestors), the aborted flag is checked first in Credential Management 1 2.5.4 Create a Credential immediately before calling [[Create]](origin, options, sameOriginWithAncestors), then in 5.1.3 Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method right before authenticator sessions start, and finally during authenticator sessions. The same goes for [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors).

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

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

5.7. Authentication Extensions (typedef AuthenticationExtensions)

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

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

5.8. Supporting Data Structures

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

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

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

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

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

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

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

The optional clientExtensions and authenticatorExtensions members contain additional parameters generated by processing the extensions passed in by the Relying Party. WebAuthn extensions are detailed in Section 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 hashAlgorithm) of the JSON-serialized client data, as constructed by the client.

4.7.2. 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".

4.7.3. Credential Descriptor (dictionary PublicKeyCredentialDescriptor)

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

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

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

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

4.7.4. Authenticator Transport enumeration (enum AuthenticatorTransport)

legitimate signature for another).

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

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

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

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

The optional clientExtensions and authenticatorExtensions members contain additional parameters generated by processing the extensions passed in by the Relying Party. WebAuthn extensions are detailed in Section 9 WebAuthn Extensions.

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

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

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

5.8.2. Credential Type enumeration (enum PublicKeyCredentialType)

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

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

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

5.8.3. Credential Descriptor (dictionary PublicKeyCredentialDescriptor)

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

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

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

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

5.8.4. Authenticator Transport enumeration (enum AuthenticatorTransport)

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

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

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

4.7.5. Cryptographic Algorithm Identifier (typedef COSEAlgorithmIdentifier)

```
typedef long COSEAlgorithmIdentifier;
```

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

5. 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


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

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

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

5.8.5. Cryptographic Algorithm Identifier (typedef COSEAlgorithmIdentifier)

```
typedef long COSEAlgorithmIdentifier;
```

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

5.8.6. User Verification Requirement enumeration (enum UserVerificationRequirement)

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

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

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

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

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

6. WebAuthn Authenticator model

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

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

For authenticators, this model defines the logical operations that they must support, and the data formats that they expose to the client and the Relying Party. However, it does not define the details of how

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

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

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

Each authenticator has an AAGUID, which is a 128-bit identifier that indicates the type (e.g. make and model) of the authenticator. The AAGUID MUST be chosen by the manufacturer to be identical across all substantially identical authenticators made by that manufacturer, and different (with probability 1-2^-128 or greater) from the AAGUIDs of all other types of authenticators. The RP MAY use the AAGUID to infer certain properties of the authenticator, such as certification level and strength of key protection, using information from other sources.

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

The goals of this design can be summarized as follows.

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

Authenticators produce cryptographic signatures for two distinct purposes:

1. An attestation signature is produced when a new public key credential is created via an authenticatorMakeCredential operation. An attestation signature provides cryptographic proof of certain properties of the the authenticator and the credential. For

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

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

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

Each authenticator has an AAGUID, which is a 128-bit identifier that indicates the type (e.g. make and model) of the authenticator. The AAGUID MUST be chosen by the manufacturer to be identical across all substantially identical authenticators made by that manufacturer, and different (with probability 1-2^-128 or greater) from the AAGUIDs of all other types of authenticators. The RP MAY use the AAGUID to infer certain properties of the authenticator, such as certification level and strength of key protection, using information from other sources.

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

The goals of this design can be summarized as follows.

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

Authenticators produce cryptographic signatures for two distinct purposes:

1. An attestation signature is produced when a new public key credential is created via an authenticatorMakeCredential operation. An attestation signature provides cryptographic proof of certain properties of the the authenticator and the credential. For

instance, an attestation signature asserts the authenticator type (as denoted by its AAGUID) and the credential public key. The attestation signature is signed by an attestation private key, which is chosen depending on the type of attestation desired. For more details on attestation, see 5.3 Attestation.

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

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

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: User Present (UP) result.
  + 1 means the user is present.
  + 0 means the user is not present.
* Bit 1: Reserved for future use (RFU1).
* Bit 2: User Verified (UV) result.
  + 1 means the user is verified.
  + 0 means the user is not verified.
* Bits 3-5: Reserved for future use (RFU2).
* Bit 6: 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.

instance, an attestation signature asserts the authenticator type (as denoted by its AAGUID) and the credential public key. The attestation signature is signed by an attestation private key, which is chosen depending on the type of attestation desired. For more details on attestation, see 6.3 Attestation.

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

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

6.1. Authenticator data

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

The authenticator data has a compact but extensible encoding. This is desired since authenticators can be devices with limited capabilities and low power requirements, with much simpler software stacks than the client platform components.

The authenticator data structure is a byte array of 37 bytes or more, as follows.

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

signCount 4 Signature counter, 32-bit unsigned big-endian integer.
attestedCredentialData variable (if present) attested credential data (if present). See 6.3.1 Attested credential data for details. Its length depends on the length of the credential ID and credential public key being attested.
extensions variable (if present) Extension-defined authenticator data. This is a CBOR [RFC7049] map with extension identifiers as keys, and authenticator extension outputs as values. See 9 WebAuthn Extensions for details.

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

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

For attestation signatures, the authenticator MUST set the AT flag and include the 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 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] 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

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

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

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

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

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

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

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

6.1.1. Signature Counter Considerations

Authenticators MUST implement a signature counter feature. The signature counter is incremented for each successful authenticatorGetAssertion operation by some positive value, and its value is returned to the Relying Party within the authenticator data. The signature counter's purpose is to aid Relying Parties in detecting cloned authenticators. Clone detection is more important for authenticators with limited protection measures.

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

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

Authenticators:
* should implement per-RP ID signature counters. This prevents the signature counter value from being shared between Relying Parties and being possibly employed as a correlation handle for the user. Authenticators may implement a global signature counter, i.e., on a per-authenticator basis, but this is less privacy-friendly for users.

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 PublicKeyCredentialUserEntity.
* A sequence of pairs of PublicKeyCredentialType and COSEAlgorithmIdentifier requested by the Relying Party. This sequence is ordered from most preferred to least preferred. The platform makes a best-effort to create the most preferred credential that it can.
* An optional list of PublicKeyCredentialDescriptor objects provided by the Relying Party with the intention that, if any of these are known to the authenticator, it should not create a new credential.
* The rk member of the options.authenticatorSelection dictionary.
* The uv member of the options.authenticatorSelection dictionary.
* Extension data created by the client based on the extensions requested by the Relying Party, if any.

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.
* should ensure that the signature counter value does not accidentally decrease (e.g., due to hardware failures).

6.2. Authenticator operations

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

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

6.2.1. The authenticatorMakeCredential operation

It takes the following input parameters:

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

rpEntity
: The Relying Party's PublicKeyCredentialRpEntity.

userEntity
: The user account's PublicKeyCredentialUserEntity, containing the user handle given by the Relying Party.

requireResidentKey
: The authenticatorSelection.requireResidentKey value given by the Relying Party.

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

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

credTypesAndPubKeyAlgs
: A sequence of pairs of PublicKeyCredentialType and public key algorithms (COSEAlgorithmIdentifier) requested by the Relying Party. This sequence is ordered from most preferred to least preferred. The platform makes a best-effort to create the most preferred credential that it can.

excludeCredentialDescriptorList
: An optional list of PublicKeyCredentialDescriptor objects provided by the Relying Party with the intention that, if any of these are known to the authenticator, it should not create a new credential. excludeCredentialDescriptorList contains a list of known credentials.

extensions
: A map from extension identifiers to their authenticator extension inputs, created by the client based on the extensions requested by the Relying Party, if any.

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

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

1. Check if all the supplied parameters are syntactically well-formed and of the correct length. If not, return an error code equivalent to "UnknownError" and terminate the operation.

1962 * Check if at least one of the specified combinations of
1963 PublicKeyCredentialType and cryptographic parameters is supported.
1964 If not, return an error code equivalent to "NotSupportedError" and
1965 terminate the operation.
1966 * Check if a credential matching any of the supplied
1967 PublicKeyCredential identifiers is present on this authenticator.
1968 If so, return an error code equivalent to "NotAllowedError" and
1969 terminate the operation.
1970 * If rk is true and the authenticator cannot store a

1971 Client-side-resident Credential Private Key, return an error code
1972 equivalent to "ConstraintError" and terminate the operation.
1973 * If uv is true and the authenticator cannot perform user
1974 verification, return an error code equivalent to "ConstraintError"
1975 and terminate the operation.
1976 * Prompt the user for consent to create a new credential. The prompt
1977 for obtaining this consent is shown by the authenticator if it has
1978 its own output capability, or by the user agent otherwise. If the
1979 user denies consent, return an error code equivalent to
1980 "NotAllowedError" and terminate the operation.
1981 * Once user consent has been obtained, generate a new credential

1982 object:
1983 + Generate a set of cryptographic keys using the most preferred
1984 combination of PublicKeyCredentialType and cryptographic
1985 parameters supported by this authenticator.
1986 + Generate an identifier for this credential, such that this
1987 identifier is globally unique with high probability across all

1988 credentials with the same type across all authenticators.
1989 + Associate the credential with the specified RP ID and the
1990 user's account identifier user.id.
1991 + Delete any older credentials with the same RP ID and user.id
1992 that are stored locally by the authenticator.
1993 * If any error occurred while creating the new credential object,

1994 return an error code equivalent to "UnknownError" and terminate the
1995 operation.
1996 * Process all the supported extensions requested by the client, and
1997 generate the authenticator data with attestation data as specified
1998 in 5.1 Authenticator data. Use this authenticator data and the
1999 hash of the serialized client data to create an attestation object
2000 for the new credential using the procedure specified in 5.3.4
2001 Generating an Attestation Object. For more details on attestation,
2002 see 5.3 Attestation.

2584 2. Check if at least one of the specified combinations of
2585 PublicKeyCredentialType and cryptographic parameters in
2586 credTypesAndPubKeyAlgs is supported. If not, return an error code
2587 equivalent to "NotSupportedError" and terminate the operation.
2588 3. Check if any credential bound to this authenticator matches an item
2589 of excludeCredentialDescriptorList. A match occurs if a credential
2590 matches rpEntity.id and an excludeCredentialDescriptorList item's
2591 excludeCredentialDescriptorList.id and
2592 excludeCredentialDescriptorList.type. If so, return an error code
2593 equivalent to "NotAllowedError" and terminate the operation.
2594 4. If requireResidentKey is true and the authenticator cannot store a
2595 Client-side-resident Credential Private Key, return an error code
2596 equivalent to "ConstraintError" and terminate the operation.
2597 5. If requireUserVerification is true and the authenticator cannot
2598 perform user verification, return an error code equivalent to
2599 "ConstraintError" and terminate the operation.
2600 6. Obtain user consent for creating a new credential. The prompt for
2601 obtaining this consent is shown by the authenticator if it has its
2602 own output capability, or by the user agent otherwise. The prompt
2603 SHOULD display rpEntity.id, rpEntity.name, userEntity.name and
2604 userEntity.displayName, if possible.
2605 If requireUserVerification is true, the method of obtaining user
2606 consent MUST include user verification.
2607 If requireUserPresence is true, the method of obtaining user
2608 consent MUST include a test of user presence.
2609 If the user denies consent or if user verification fails, return an
2610 error code equivalent to "NotAllowedError" and terminate the
2611 operation.
2612 7. Once user consent has been obtained, generate a new credential
2613 object:
2614 1. Let (publicKey,privateKey) be a new pair of cryptographic keys
2615 using the combination of PublicKeyCredentialType and
2616 cryptographic parameters represented by the first item in
2617 credTypesAndPubKeyAlgs that is supported by this
2618 authenticator.
2619 2. Let credentialId be a new identifier for this credential that
2620 is globally unique with high probability across all
2621 credentials with the same type across all authenticators.
2622 3. Let userHandle be userEntity.id.
2623 4. Associate the credentialId and privateKey with rpEntity.id and
2624 userHandle.
2625 5. Delete any older credentials with the same rpEntity.id and
2626 userHandle that are stored locally by the authenticator.
2627 8. If any error occurred while creating the new credential object,
2628 return an error code equivalent to "UnknownError" and terminate the
2629 operation.
2630 9. Let processedExtensions be the result of authenticator extension
2631 processing for each supported extension identifier/input pair in
2632 extensions.
2633 10. If the authenticator supports:
2634
2635 a per-RP ID signature counter
2636 allocate the counter, associate it with the RP ID, and
2637 initialize the counter value as zero.
2638
2639 a global signature counter
2640 Use the global signature counter's actual value when
2641 generating authenticator data.
2642
2643 a per credential signature counter
2644 allocate the counter, associate it with the new
2645 credential, and initialize the counter value as zero.
2646
2647 11. Let attestedCredentialData be the attested credential data byte
2648 array including the credentialId and publicKey.
2649 12. Let authenticatorData be the byte array specified in 6.1
2650 Authenticator data, including attestedCredentialData as the
2651 attestedCredentialData and processedExtensions, if any, as the
2652 extensions.
2653 13. Return the attestation object for the new credential created by the

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), if any.
* Extension data created by the client based on the extensions requested by the Relying Party, if any.

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

* 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

procedure specified in 6.3.4 Generating an Attestation Object using an authenticator-chosen attestation statement format, authenticatorData, and hash. For more details on attestation, see 6.3 Attestation.

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

6.2.2. The authenticatorGetAssertion operation

It takes the following input parameters:

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

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

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

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

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

extensions
: A map from extension identifiers to their authenticator extension inputs, created by the client based on the extensions requested by the Relying Party, if any.

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

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

1. Check if all the supplied parameters are syntactically well-formed and of the correct length. If not, return an error code equivalent to "UnknownError" and terminate the operation.
2. If requireUserVerification is true and the authenticator cannot perform user verification, return an error code equivalent to "ConstraintError" and terminate the operation.
3. If allowCredentialDescriptorList was not supplied, set it to a list of all credentials stored for rpId (as determined by an exact match of rpId).
4. Remove any items from allowCredentialDescriptorList that do not match a credential bound to this authenticator. A match occurs if a credential matches rpId and an allowCredentialDescriptorList item's id and type members.
5. If allowCredentialDescriptorList is now empty, return an error code equivalent to "NotAllowedError" and terminate the operation.
6. Let selectedCredential be a credential as follows. If the size of allowCredentialDescriptorList

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

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

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 as specified in 5.1 Authenticator data, though without attestation 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 in Figure 2, below. A simple,

undelimited concatenation is safe to use here because the authenticator data describes its own length. The hash of the serialized client data (which potentially has a variable length) is always the last element.

* 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 an assertion signature.

On successful completion, the authenticator returns to the user agent:

* The identifier of the credential (credential ID) used to generate the assertion signature.
* The authenticator data used to generate the assertion 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

allowCredentialDescriptorList.

7. Obtain user consent for using selectedCredential. The prompt for obtaining this consent may be shown by the authenticator if it has its own output capability, or by the user agent otherwise. The prompt SHOULD display the rpId and any additional displayable data associated with selectedCredential, if possible.
   If requireUserVerification is true, the method of obtaining user consent MUST include user verification.
   If requireUserPresence is true, the method of obtaining user consent MUST include a test of user presence.
   If the user denies consent or if user verification fails, return an error code equivalent to "NotAllowedError" and terminate the operation.
8. Let processedExtensions be the result of authenticator extension processing for each supported extension identifier/input pair in extensions.
9. Increment the RP ID-associated signature counter or the global signature counter value, depending on which approach is implemented by the authenticator, by some positive value.
10. Let authenticatorData be the byte array specified in 6.1 Authenticator data including processedExtensions, if any, as the extensions and excluding attestedCredentialData.
11. Let signature be the assertion signature of the concatenation authenticatorData || hash using the private key of selectedCredential as shown in Figure 2, below. A simple,

undelimited concatenation is safe to use here because the authenticator data describes its own length. The hash of the serialized client data (which potentially has a variable length) is always the last element.

Generating an assertion signature Generating an assertion signature.

12. If any error occurred while generating the assertion signature,

return an error code equivalent to "UnknownError" and terminate the operation.

13. Return to the user agent:
    + selectedCredential's credential ID, if either a list of credentials of size 2 or greater was supplied by the client, or no such list was supplied. Otherwise, return only the below values.
      Note: If the client supplies a list of exactly one credential and it was successfully employed, then its credential ID is not returned since the client already knows it. This saves transmitting these bytes over what may be a constrained connection in what is likely a common case.
    + authenticatorData
    + signature
    + The user handle associated with selectedCredential.

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

6.2.3. The authenticatorCancel operation

This operation takes no input parameters and returns no result.

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

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

authenticatorGetAssertion operation currently in progress.

5.3. Attestation

Authenticators must also provide some form of attestation. The basic requirement is that the authenticator can produce, for each credential public key, an attestation statement verifable by the Relying Party. Typically, this attestation statement contains a signature by an attestation private key over the attested credential public key and a challenge, as well as a certificate or similar data providing provenance information for the attestation public key, enabling the Relying Party to make a trust decision. However, if an attestation key pair is not available, then the authenticator MUST perform self attestation of the credential public key with the corresponding credential private key. All this information is returned by authenticators any time a new public key credential is generated, in the overall form of an attestation object. The relationship of the attestation object with authenticator data (containing attestation data) and the attestation statement is illustrated in figure 3, below.

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

This figure illustrates only the packed attestation statement format. Several additional attestation statement formats are defined in 7 Defined Attestation Statement Formats.

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

1. The attestation statement format is the manner in which the signature is represented and the various contextual bindings are incorporated into the attestation statement by the authenticator. In other words, this defines the syntax of the statement. Various existing devices and platforms (such as TPMs and the Android OS) have previously defined attestation statement formats. This specification supports a variety of such formats in an extensible way, as defined in 5.3.2 Attestation Statement Formats.
2. The attestation type defines the semantics of attestation statements and their underlying trust models. Specifically, it defines how a Relying Party establishes trust in a particular attestation statement, after verifying that it is cryptographically valid. This specification supports a number of attestation types, as described in 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

authenticatorGetAssertion operation currently in progress.

6.3. Attestation

Authenticators must also provide some form of attestation. The basic requirement is that the authenticator can produce, for each credential public key, an attestation statement verifable by the Relying Party. Typically, this attestation statement contains a signature by an attestation private key over the attested credential public key and a challenge, as well as a certificate or similar data providing provenance information for the attestation public key, enabling the Relying Party to make a trust decision. However, if an attestation key pair is not available, then the authenticator MUST perform self attestation of the credential public key with the corresponding credential private key. All this information is returned by authenticators any time a new public key credential is generated, in the overall form of an attestation object. The relationship of the attestation object with authenticator data (containing attested credential data) and the attestation statement is illustrated in figure 3, below.

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

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

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

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

1. The attestation statement format is the manner in which the signature is represented and the various contextual bindings are incorporated into the attestation statement by the authenticator. In other words, this defines the syntax of the statement. Various existing devices and platforms (such as TPMs and the Android OS) have previously defined attestation statement formats. This specification supports a variety of such formats in an extensible way, as defined in 6.3.2 Attestation Statement Formats.
2. The attestation type defines the semantics of attestation statements and their underlying trust models. Specifically, it defines how a Relying Party establishes trust in a particular attestation statement, after verifying that it is cryptographically valid. This specification supports a number of attestation types, as described in 6.3.3 Attestation Types.

In general, there is no simple mapping between attestation statement formats and attestation types. For example, the "packed" attestation statement format defined in 8.2 Packed Attestation Statement Format can be used in conjunction with all attestation types, while other 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 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 The credential public key encoded in COSE_Key format, as defined in Section 7 of [RFC8152]. The encoded credential public key MUST contain the "alg" parameter and MUST NOT contain any other optional parameters. The "alg" parameter MUST contain a COSEAlgorithmIdentifier value.

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 MUST be defined using the following template:

* Attestation statement format identifier:
* Supported attestation types:
* Syntax: The syntax of an attestation statement produced in this format, defined using [CDDL] for the extension point $attStmtFormat defined in 5.3.4 Generating an Attestation Object.
* Signing procedure: The signing procedure for computing an attestation statement in this format given the public key credential to be attested, the authenticator data structure containing the authenticator data for the attestation, and the hash of the serialized client data.
* Verification procedures: 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 specified 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.

Parties will decide what attestation types are acceptable to them by policy. Relying Parties will also need to understand the characteristics of the authenticators that they trust, based on information they have about these authenticators. For example, the FIDO Metadata Service [FIDOMetadataService] provides one way to access such information.

6.3.1. Attested credential data

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

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

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

6.3.2. Attestation Statement Formats

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

* Attestation statement format identifier:
* Supported attestation types:
* Syntax: The syntax of an attestation statement produced in this format, defined using [CDDL] for the extension point $attStmtFormat defined in 6.3.4 Generating an Attestation Object.
* Signing procedure: The signing procedure for computing an attestation statement in this format given the public key credential to be attested, the authenticator data structure containing the authenticator data for the attestation, and the hash of the serialized client data.
* Verification procedure: The procedure for verifying an attestation statement, which takes the following verification procedure inputs:
    + attStmt: The attestation statement structure
    + authenticatorData: The authenticator data claimed to have been used for the attestation
    + clientDataHash: The hash of the serialized client data

  The procedure returns either:
    + An error indicating that the attestation is invalid, or
    + The attestation type, and the trust path. This attestation trust path is either empty (in case of self attestation), an identifier of a ECDAA-Issuer public key (in the case of ECDAA), or a set of X.509 certificates.

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

6.3.3. Attestation Types

WebAuthn supports multiple attestation types:

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

Self Attestation

In the case of self attestation, also known as surrogate basic attestation [UAFProtocol], the Authenticator does not have any specific attestation key. Instead it uses the 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 ECDAA-Issuer (see [FIDOEcdaaAlgorithm]).

5.3.4. Generating an Attestation Object

This section specifies the algorithm for generating an attestation object (see: Figure 3) for any attestation statement format.

In order to construct an attestation object for a given public key 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
```


Self Attestation

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

Privacy CA

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

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

Elliptic Curve based Direct Anonymous Attestation (ECDAA)

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

6.3.4. Generating an Attestation Object

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

attestationFormat
: An attestation statement format.

authData
: A byte array containing authenticator data.

hash
: The hash of the serialized client data.

the authenticator MUST:

1. Let attStmt be the result of running attestationFormat's signing procedure given authData and hash.
2. Let fmt be attestationFormat's attestation statement format identifier
3. Return the attestation object as a CBOR map with the following syntax, filled in with variables initialized by this algorithm:

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

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

; Every attestation statement format must have the above fields
```

attStmtTemplate .within $$attStmtType

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 scheme, the authenticator generates a blinded attestation signature. This allows the Relying Party to verify the signature using the ECDAA-Issuer public key, but the attestation signature does not serve as a global correlation handle.

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

attStmtTemplate .within $$attStmtType

6.3.5. Security Considerations

6.3.5.1. Privacy

Attestation keys may be used to track users or link various online identities of the same user together. This may be mitigated in several ways, including:

* A WebAuthn authenticator manufacturer may choose to ship all of their devices with the same (or a fixed number of) attestation key(s) (called Basic Attestation). This will anonymize the user at the risk of not being able to revoke a particular attestation key should its WebAuthn Authenticator be compromised.
* A WebAuthn Authenticator may be capable of dynamically generating different attestation keys (and requesting related certificates) per origin (following the Privacy CA approach). For example, a WebAuthn Authenticator can ship with a master attestation key (and certificate), and combined with a cloud operated privacy CA, can dynamically generate per origin attestation keys and attestation certificates.
* A WebAuthn Authenticator can implement Elliptic Curve based direct anonymous attestation (see [FIDOEcdaaAlgorithm]). Using this scheme, the authenticator generates a blinded attestation signature. This allows the Relying Party to verify the signature using the ECDAA-Issuer public key, but the attestation signature does not serve as a global correlation handle.

6.3.5.2. Attestation Certificate and Attestation Certificate CA Compromise

When an intermediate CA or a root CA used for issuing attestation certificates is compromised, WebAuthn authenticator attestation keys are still safe although their certificates can no longer be trusted. A WebAuthn Authenticator manufacturer that has recorded the public attestation keys for their devices can issue new attestation certificates for these keys from a new intermediate CA or from a new root CA. If the root CA changes, the Relying Parties must update their trusted root certificates accordingly.

A WebAuthn Authenticator attestation certificate must be revoked by the issuing CA if its key has been compromised. A WebAuthn Authenticator manufacturer may need to ship a firmware update and inject new attestation keys and certificates into already manufactured WebAuthn Authenticators, if the exposure was due to a firmware flaw. (The process by which this happens is out of scope for this specification.) If the WebAuthn Authenticator manufacturer does not have this capability, then it may not be possible for Relying Parties to trust any further attestation statements from the affected WebAuthn Authenticators.

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 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 tokenBindingId 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.hashAlgorithm.
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].

attestation") public key credentials that were registered after the CA compromise date using an attestation certificate chaining up to the same intermediate CA. It is thus recommended that Relying Parties remember intermediate attestation CA certificates during Authenticator registration in order to un-register related public key credentials if the registration was performed after revocation of such certificates.

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

6.3.5.3. Attestation Certificate Hierarchy

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

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

7. Relying Party Operations

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

7.1. Registering a new credential

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

1. Perform JSON deserialization on the clientDataJSON field of the AuthenticatorAttestationResponse object to extract the client data C claimed as collected during the credential creation.
2. Verify that the type in C is the string webauthn.create.
3. Verify that the challenge in C matches the challenge that was sent to the authenticator in the create() call.
4. Verify that the origin in C matches the Relying Party's origin.
5. Verify that the tokenBindingId in C matches the Token Binding ID for the TLS connection over which the attestation was obtained.
6. Verify that the clientExtensions in C is a subset of the extensions requested by the RP and that the authenticatorExtensions in C is also a subset of the extensions requested by the RP.
7. Compute the hash of clientDataJSON using the algorithm identified by C.hashAlgorithm.
8. Perform CBOR decoding on the attestationObject field of the AuthenticatorAttestationResponse structure to obtain the attestation statement format fmt, the authenticator data authData, and the attestation statement attStmt.
9. Verify that the RP ID hash in authData is indeed the SHA-256 hash of the RP ID expected by the RP.
10. Determine the attestation statement format by performing an USASCII case-sensitive match on fmt against the set of supported WebAuthn Attestation Statement Format Identifier values. The up-to-date list of registered WebAuthn Attestation Statement Format Identifier values is maintained in the in the IANA registry of the same name [WebAuthn-Registries].

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 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.

11. Verify that attStmt is a correct attestation statement, conveying a valid attestation signature, by using the attestation statement format fmt's verification procedure given attStmt, authData and the hash of the serialized client data computed in step 6.
    Note: Each attestation statement format specifies its own verification procedure. See 8 Defined Attestation Statement Formats for the initially-defined formats, and [WebAuthn-Registries] for the up-to-date list.
12. If validation is successful, obtain a list of acceptable trust anchors (attestation root certificates or ECDAA-Issuer public keys) for that attestation type and attestation statement format fmt, from a trusted source or from policy. For example, the FIDO Metadata Service [FIDOMetadataService] provides one way to obtain such information, using the aaguid in the attestedCredentialData in authData.
13. Assess the attestation trustworthiness using the outputs of the verification procedure in step 10, as follows:
    + If self attestation was used, check if self attestation is acceptable under Relying Party policy.
    + If ECDAA was used, verify that the identifier of the ECDAA-Issuer public key used is included in the set of acceptable trust anchors obtained in step 11.
    + Otherwise, use the X.509 certificates returned by the verification procedure to verify that the attestation public key correctly chains up to an acceptable root certificate.
14. If the attestation statement attStmt verified successfully and is found to be trustworthy, then register the new credential with the account that was denoted in the options.user passed to create(), by associating it with the credentialId and credentialPublicKey in the attestedCredentialData in authData, as appropriate for the Relying Party's system.
15. If the attestation statement attStmt successfully verified but is not trustworthy per step 12 above, the Relying Party SHOULD fail the registration ceremony.
    NOTE: However, if permitted by policy, the Relying Party MAY register the credential ID and credential public key but treat the credential as one with self attestation (see 6.3.3 Attestation Types). If doing so, the Relying Party is asserting there is no cryptographic proof that the public key credential has been generated by a particular authenticator model. See [FIDOSecRef] and [UAFProtocol] for a more detailed discussion.

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

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

7.2. Verifying an authentication assertion

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

1. Using credential's id attribute (or the corresponding rawId, if base64url encoding is inappropriate for your use case), look up the corresponding credential public key.
2. Let cData, aData and sig denote the value of credential's response's clientDataJSON, authenticatorData, and signature respectively.

3. Perform JSON deserialization on cData to extract the client data C used for the signature.
4. Verify that the 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 tokenBindingId 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 hashAlgorithm 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 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,

3. Perform JSON deserialization on cData to extract the client data C used for the signature.
4. Verify that the type in C is the string webauthn.get.
5. Verify that the challenge member of C matches the challenge that was sent to the authenticator in the PublicKeyCredentialRequestOptions passed to the get() call.
6. Verify that the origin member of C matches the Relying Party's origin.
7. Verify that the tokenBindingId member of C (if present) matches the Token Binding ID for the TLS connection over which the signature was obtained.
8. Verify that the clientExtensions member of C is a subset of the extensions requested by the Relying Party and that the authenticatorExtensions in C is also a subset of the extensions requested by the Relying Party.
9. Verify that the rpIdHash in aData is the SHA-256 hash of the RP ID expected by the Relying Party.
10. Let hash be the result of computing a hash over the cData using the algorithm represented by the hashAlgorithm member of C.
11. Using the credential public key looked up in step 1, verify that sig is a valid signature over the binary concatenation of aData and hash.
12. If the signature counter value adata.signCount is nonzero or the value stored in conjunction with credential's id attribute is nonzero, then run the following substep:
    + If the signature counter value adata.signCount is

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

        less than or equal to the signature counter value stored in conjunction with credential's id attribute.
        This is an signal that the authenticator may be cloned, i.e. at least two copies of the credential private key may exist and are being used in parallel. Relying Parties should incorporate this information into their risk scoring. Whether the Relying Party updates the stored signature counter value in this case, or not, or fails the authentication ceremony or not, is Relying Party-specific.

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

8. Defined Attestation Statement Formats

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

8.1. Attestation Statement Format Identifiers

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

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

Unregistered attestation statement format identifiers SHOULD use lowercase reverse domain-name naming, using a domain name registered by the developer, in order to assure uniqueness of the identifier. All attestation statement format identifiers MUST be a maximum of 32 octets in length and MUST consist only of printable USASCII characters,

excluding backslash and doublequote, i.e., VCHAR as defined in [RFC5234] but without %x22 and %x5c.

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

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

Attestation statement formats that may exist in multiple versions SHOULD include a version in their identifier. In effect, different versions are thus treated as different formats, e.g., packed2 as a new version of the packed attestation statement format.

The following sections present a set of currently-defined and registered attestation statement formats and their identifiers. The up-to-date list of registered WebAuthn Extensions is maintained in the IANA "WebAuthn Attestation Statement Format Identifier" registry established by [WebAuthn-Registries].

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.

excluding backslash and doublequote, i.e., VCHAR as defined in [RFC5234] but without %x22 and %x5c.

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

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

Attestation statement formats that may exist in multiple versions SHOULD include a version in their identifier. In effect, different versions are thus treated as different formats, e.g., packed2 as a new version of the packed attestation statement format.

The following sections present a set of currently-defined and registered attestation statement formats and their identifiers. The up-to-date list of registered WebAuthn Extensions is maintained in the IANA "WebAuthn Attestation Statement Format Identifier" registry established by [WebAuthn-Registries].

8.2. Packed Attestation Statement Format

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

Attestation statement format identifier
: packed

Attestation types supported
: All

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

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

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

The semantics of the fields are as follows:

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

sig
: A byte string containing the attestation signature.

x5c
: The elements of this array contain the attestation certificate and its certificate chain, each encoded in X.509 format. The attestation certificate must be the first element in the array.

ecdaaKeyId
The identifier of the ECDAA-Issuer public key. This is the BigNumberToB encoding of the component "c" of the 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 authenticatorData and clientDataHash using ECDAA-Verify with ECDAA-Issuer public key identified by ecdaaKeyId (see [FIDOEcdaaAlgorithm]).

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.

1. Let authenticatorData denote the authenticator data for the attestation, and let clientDataHash denote the hash of the serialized client data.
2. If Basic or Privacy CA attestation is in use, the authenticator produces the sig by concatenating authenticatorData and clientDataHash, and signing the result using an attestation private key selected through an authenticator-specific mechanism. It sets x5c to the certificate chain of the attestation public key and alg to the algorithm of the attestation private key.
3. 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]) after selecting an ECDAA-Issuer public key related to the ECDAA signature private key through an authenticator-specific mechanism (see [FIDOEcdaaAlgorithm]). It sets alg to the algorithm of the selected ECDAA-Issuer public key and ecdaaKeyId to the identifier of the ECDAA-Issuer public key (see above).
4. If self attestation is in use, the authenticator produces sig by concatenating authenticatorData and clientDataHash, and signing the result using the credential private key. It sets alg to the algorithm of the credential private key, and omits the other fields.

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

1. Verify that attStmt is valid CBOR conforming to the syntax defined above, and perform CBOR decoding on it to extract the contained fields.
2. If x5c is present, this indicates that the attestation type is not ECDAA. In this case:
   - o 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.
   - o Verify that x5c meets the requirements in 8.2.1 Packed attestation statement certificate requirements.
   - o If x5c contains an extension with OID 1 3 6 1 4 1 45724 1 1 4 (id-fido-gen-ce-aaguid) verify that the value of this extension matches the aaguid in authenticatorData.
   - o If successful, return attestation type Basic and attestation trust path x5c.
3. If ecdaaKeyId is present, then the attestation type is ECDAA. In this case:
   - o Verify that sig is a valid signature over the concatenation of authenticatorData and clientDataHash using ECDAA-Verify with ECDAA-Issuer public key identified by ecdaaKeyId (see [FIDOEcdaaAlgorithm]).
   - o If successful, return attestation type ECDAA and attestation trust path ecdaaKeyId.

```
+ 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.
```

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

7.2.1. Packed attestation statement certificate requirements

8.2.1. Packed attestation statement certificate requirements

The attestation certificate MUST have the following fields/extensions:

* Version must be set to 3.
* Subject field MUST be set to:

  Subject-C
  : Country where the Authenticator vendor is incorporated

  Subject-O
  : Legal name of the Authenticator vendor

  Subject-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

8.3. 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) ]
                    ) //
                    (
```

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

tpmStmtFormat = {
                    ver: "2.0",
                    (
                        alg: COSEAlgorithmIdentifier,
                        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.

```
                    alg: COSEAlgorithmIdentifier, (-260 for ED256 / -26
1 for ED512)
                    ecdaaKeyId: bytes
                ),
                sig: bytes,
                certInfo: bytes,
                pubArea: bytes
            }
```

The semantics of the above fields are as follows:

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

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

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

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

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

certInfo
: The TPMS_ATTEST structure over which the above signature was computed, as specified in [TPMv2-Part2] section 10.12.8.

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

Signing procedure

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

Concatenate authenticatorData and clientDataHash to form attToBeSigned.

Generate a signature using the procedure specified in [TPMv2-Part3] Section 18.2, using the attestation private key and setting the extraData parameter to the digest of attToBeSigned using the hash algorithm corresponding to the "alg" signature algorithm. (For the "RS256" algorithm, this would be a SHA-256 digest.)

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

Verification procedure

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

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 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

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

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

Concatenate authenticatorData and clientDataHash to form attToBeSigned.

Validate that certInfo is valid:

+ Verify that magic is set to TPM_GENERATED_VALUE.
+ Verify that type is set to TPM_ST_ATTEST_CERTIFY.
+ Verify that extraData is set to the hash of attToBeSigned using the hash algorithm employed in "alg".
+ Verify that attested contains a TPMS_CERTIFY_INFO structure, whose name field contains a valid Name for pubArea, as computed using the algorithm in the nameAlg field of pubArea using the procedure specified in [TPMv2-Part1] section 16.

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

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

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

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

8.3.1. TPM attestation statement certificate requirements

TPM attestation certificate MUST have the following fields/extensions:

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

8.4. Android Key Attestation Statement Format

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

consistent with the fields of the attestation certificate's extension data.

Attestation statement format identifier
android-key

Attestation types supported
Basic

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

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

androidStmtFormat = 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 an 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 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.

consistent with the fields of the attestation certificate's extension data.

Attestation statement format identifier
android-key

Attestation types supported
Basic Attestation

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

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

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

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

Request an Android Key Attestation by calling "keyStore.getCertificateChain(myKeyUUID)") providing clientDataHash as the challenge value (e.g., by using setAttestationChallenge). Set x5c to the returned value.

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

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

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

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

statement.

8.5. Android SafetyNet Attestation Statement Format

When the authenticator in question is a platform-provided Authenticator on certain Android platforms, the attestation statement is based on the SafetyNet API. In this case the authenticator data is completely controlled by the caller of the SafetyNet API (typically an application running on the Android platform) and the attestation statement only provides some statements about the health of the platform and the identity of the calling application. This attestation does not provide information regarding provenance of the authenticator and its associated data. Therefore platform-provided authenticators should make use of the Android Key Attestation when available, even if the SafetyNet API is also present.

Attestation statement format identifier
android-safetynet

Attestation types supported
Basic Attestation

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

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

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

The semantics of the above fields are as follows:

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

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

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

Concatenate authenticatorData and clientDataHash to form attToBeSigned.

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

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

+ Verify that attStmt is valid CBOR conforming to the syntax defined above, and perform CBOR decoding on it to extract the contained fields.
+ Verify that response is a valid SafetyNet response of version ver.
+ Verify that the nonce in the response is identical to the

concatenation of 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, self attestation

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 -7 ("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.

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

8.6. FIDO U2F Attestation Statement Format

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

Attestation statement format identifier
fido-u2f

Attestation types supported
Basic Attestation, Self Attestation, Privacy CA

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

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

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

The semantics of the above fields are as follows:

x5c
The elements of this array contain the attestation certificate and its certificate chain, each encoded in X.509 format. The attestation certificate must be the first element in the array.

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

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

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

Generate a Registration Response Message as specified in [FIDO-U2F-Message-Formats] section 4.3, with the application parameter set to the SHA-256 hash of the RP ID associated with the given credential, the challenge parameter set to tbsHash, and the key handle parameter set to the credential ID of the given credential. Set the raw signature part of this Registration Response Message (i.e., without the user public key, key handle, and attestation certificates) as sig and set the attestation certificates of the attestation public key as x5c.

Verification procedure

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

Verification procedure

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

1. Verify that attStmt is valid CBOR conforming to the syntax defined above, and perform CBOR decoding on it to extract the contained fields.
2. Let attCert be value of the first element of x5c. Let certificate public key be the public key conveyed by attCert. If certificate public key is not an Elliptic Curve (EC) public key over the P-256 curve, terminate this algorithm and return an appropriate error.
3. Extract the claimed rpIdHash from authenticatorData, and the claimed credentialId and credentialPublicKey from authenticatorData.attestedCredentialData.
4. If clientDataHash is 256 bits long, set tbsHash to this value.
   Otherwise set tbsHash to the SHA-256 hash of clientDataHash.
5. Convert the COSE_KEY formatted credentialPublicKey (see Section 7 of [RFC8152]) to CTAP1/U2F public Key format [FIDO-CTAP].
   - o Let publicKeyU2F represent the result of the conversion operation and set its first byte to 0x04. Note: This signifies uncompressed ECC key format.
   - o Extract the value corresponding to the "-2" key (representing x coordinate) from credentialPublicKey, confirm its size to be of 32 bytes and concatenate it with publicKeyU2F. If size differs or "-2" key is not found, terminate this algorithm and return an appropriate error.
   - o Extract the value corresponding to the "-3" key (representing y coordinate) from credentialPublicKey, confirm its size to be of 32 bytes and concatenate it with publicKeyU2F. If size differs or "-3" key is not found, terminate this algorithm and return an appropriate error.
6. Let verificationData be the concatenation of (0x00 || rpIdHash || tbsHash || credentialId || publicKeyU2F) (see Section 4.3 of [FIDO-U2F-Message-Formats]).
7. Verify the sig using verificationData and certificate public key per [SEC1].
8. If successful, return attestation type Basic with the attestation trust path set to x5c.

9. WebAuthn Extensions

The mechanism for generating public key credentials, as well as requesting and generating Authentication assertions, as defined in 5 Web Authentication API, can be extended to suit particular use cases. Each case is addressed by defining a registration extension and/or an authentication extension.

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

* navigator.credentials.create() extension request parameters and response values for registration extensions.
* navigator.credentials.get() extension request parameters and response values for authentication extensions.
* Client extension processing for registration extensions and authentication extensions.

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

extension processing for each extension that it supports, and augments the client data as specified by each extension, by including the extension identifier and client extension output values.

An extension can also be an authenticator extension, meaning that the extension invoves communication with and processing by the authenticator. Authenticator extensions define the following steps and data:

* authenticatorMakeCredential extension request parameters and response values for registration extensions.
* authenticatorGetAssertion extension request parameters and response values for authentication extensions.
* Authenticator extension processing for registration extensions and authentication extensions.

For authenticator extensions, as part of the client extension 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,

extension processing for each extension that it supports, and augments the client data as specified by each extension, by including the extension identifier and client extension output values.

An extension can also be an authenticator extension, meaning that the extension invoves communication with and processing by the authenticator. Authenticator extensions define the following steps and data:

* authenticatorMakeCredential extension request parameters and response values for registration extensions.
* authenticatorGetAssertion extension request parameters and response values for authentication extensions.
* Authenticator extension processing for registration extensions and authentication extensions.

For authenticator extensions, as part of the client extension 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.

9.1. Extension Identifiers

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

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

Unregistered extension identifiers should aim to be globally unique,

e.g., by including the defining entity such as myCompany_extension.

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

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

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

e.g., by including the defining entity such as myCompany_extension.

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

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

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

9.2. Defining extensions

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

Any client extension that is processed by the client MUST return a client extension output value so that the Relying Party knows that the extension was honored by the client. Similarly, any extension that requires authenticator processing MUST return an authenticator extension output to let the Relying Party know that the extension was honored by the authenticator. If an extension does not otherwise require any result values, it SHOULD be defined as returning a JSON Boolean client extension output result, set to true to signify that the extension was understood and processed. Likewise, any authenticator extension that does not otherwise require any result values MUST return a value and SHOULD return a CBOR Boolean authenticator extension output result, set to true to signify that the extension was understood and processed.

9.3. Extending request parameters

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

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

```
var assertionPromise = navigator.credentials.get({
    publicKey: {
        // The challenge must be produced by the server, see the Security Considerations
        challenge: new Uint8Array([4,99,22 /* 29 more random bytes generated by the server */]),
        extensions: {
            "webauthnExample_foobar": 42
        }
    }
});
```

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

client extension input. If an extension does not require any parameters from the Relying Party, it SHOULD be defined as taking a Boolean client argument, set to true to signify that the extension is requested by the Relying Party.

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

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

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 extension input. If an extension does not require any parameters from the Relying Party, it SHOULD be defined as taking a Boolean client argument, set to true to signify that the extension is requested by the Relying Party.

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

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

9.4. Client extension processing

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

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

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

9.5. Authenticator extension processing

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

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

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

9.6. Example Extension

This section is not normative.

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

The extension identifier is chosen as webauthnExample_geo. The client extension input is the constant value true, since the extension does not require the Relying Party to pass any particular information to the

```
    client, other than that it requests the use of the extension. The
    Relying Party sets this value in its request for an assertion:
var assertionPromise =
    navigator.credentials.get({
        publicKey: {
            challenge: "SGFulFNvbG8gc2hvdCBmaXJzdC4",
            allowCredentials: [], /* Empty filter */
            extensions: { 'webauthnExample_geo': true }
        }
    });

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

    The extension requires the authenticator to specify its geolocation in
    the authenticator extension output, if known. The extension e.g.
    specifies that the location shall be encoded as a two-element array of
    floating point numbers, encoded with CBOR. An authenticator does this
    by including it in the authenticator data. As an example, authenticator
    data may be as follows (notation taken from [RFC7049]):
81 (hex)                        -- Flags, ED and UP both set.
20 05 58 1F                       -- Signature counter
A1                              -- CBOR map of one element
   73                           -- Key 1: CBOR text string of 19 byt
es
      77 65 62 61 75 74 68 6E 45 78 61
      6D 70 6C 65 5F 67 65 6F          -- "webauthnExample_geo" [=UTF-8 enc
oded=] string
   82                           -- Value 1: CBOR array of two elemen
ts
      FA 42 82 1E B3                -- Element 1: Latitude as CBOR encod
ed float
      FA C1 5F E3 7F                -- Element 2: Longitude as CBOR enco
ded float

    The extension defines the client extension output to be the geolocation
    information, if known, as a GeoJSON [GeoJSON] point. The client
    constructs the following client data:
{
    ...,
    'extensions': {
        'webauthnExample_geo': {
            'type': 'Point',
            'coordinates': [65.059962, -13.993041]
        }
    }
}

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

    Extension identifier
        appid
```

```
    client, other than that it requests the use of the extension. The
    Relying Party sets this value in its request for an assertion:
var assertionPromise =
    navigator.credentials.get({
        publicKey: {
            // The challenge must be produced by the server, see the Security Co
nsiderations
            challenge: new Uint8Array([11,103,35 /* 29 more random bytes generat
ed by the server */]),
            allowCredentials: [], /* Empty filter */
            extensions: { 'webauthnExample_geo': true }
        }
    });

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

    The extension requires the authenticator to specify its geolocation in
    the authenticator extension output, if known. The extension e.g.
    specifies that the location shall be encoded as a two-element array of
    floating point numbers, encoded with CBOR. An authenticator does this
    by including it in the authenticator data. As an example, authenticator
    data may be as follows (notation taken from [RFC7049]):
81 (hex)                        -- Flags, ED and UP both set.
20 05 58 1F                       -- Signature counter
A1                              -- CBOR map of one element
   73                           -- Key 1: CBOR text string of 19 byt
es
      77 65 62 61 75 74 68 6E 45 78 61
      6D 70 6C 65 5F 67 65 6F          -- "webauthnExample_geo" [=UTF-8 enc
oded=] string
   82                           -- Value 1: CBOR array of two elemen
ts
      FA 42 82 1E B3                -- Element 1: Latitude as CBOR encod
ed float
      FA C1 5F E3 7F                -- Element 2: Longitude as CBOR enco
ded float

    The extension defines the client extension output to be the geolocation
    information, if known, as a GeoJSON [GeoJSON] point. The client
    constructs the following client data:
{
    ...,
    'extensions': {
        'webauthnExample_geo': {
            'type': 'Point',
            'coordinates': [65.059962, -13.993041]
        }
    }
}

10. Defined Extensions

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

  10.1. FIDO AppId Extension (appid)

    This authentication extension allows Relying Parties that have
    previously registered a credential using the legacy FIDO JavaScript
    APIs to request an assertion. Specifically, this extension allows
    Relying Parties to specify an appId [FIDO-APPID] to overwrite the
    otherwise computed rpId. This extension is only valid if used during
    the get() call; other usage will result in client error.

    Extension identifier
        appid
```

Client extension input
A single JSON string specifying a FIDO appId.

Client extension processing
If rpId is present, 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 to make an assertion - PublicKeyCredential's [[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 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.

Client extension input
A single JSON string specifying a FIDO appId.

Client extension processing
If rpId is present, return a DOMException whose name is "NotAllowedError", and terminate this algorithm (5.1.4.1 PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method).

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

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

Authenticator extension input
None.

Authenticator extension processing
None.

Authenticator extension output
None.

10.2. Simple Transaction Authorization Extension (txAuthSimple)

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

Extension identifier
txAuthSimple

Client extension input
A single JSON string prompt.

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

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

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

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

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

10.3. Generic Transaction Authorization Extension (txAuthGeneric)

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

Extension identifier
txAuthGeneric

Client extension input
A CBOR map defined as follows:

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

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

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.

10.4. Authenticator Selection Extension (authnSel)

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

Extension identifier
authnSel

Client extension input
A sequence of AAGUIDs:

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

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

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

```
typedef BufferSource      AAGUID;
```

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

Client extension output

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

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 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

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

Authenticator extension input
None.

Authenticator extension processing
None.

Authenticator extension output
None.

10.5. Supported Extensions Extension (exts)

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

Extension identifier
exts

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

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

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

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

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

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

10.6. User Verification Index Extension (uvi)

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

Extension identifier
uvi

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

Client extension processing
None, except creating the authenticator extension input from the 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 authenticator data containing one UVI extension

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

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.

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

Authenticator extension output

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

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

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

Servers supporting UVI extensions MUST support a length of up to 32 bytes for the UVI value.

Example for authenticator data containing one UVI extension

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

10.7. Location Extension (loc)

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

Extension identifier

loc

Client extension input

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

Client extension processing

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

Client extension output

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

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

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

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

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

9.8. User Verification Method Extension (uvm)

10.8. User Verification Method Extension (uvm)

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

Extension identifier
uvm

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

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

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

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

Authenticator extension processing
The authenticator sets the authenticator extension output to be 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 authenticator data containing one UVM extension for a multi-factor authentication instance where 2 factors were used:

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

ftware

10. IANA Considerations

10.1. WebAuthn Attestation Statement Format Identifier Registrations

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 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

ftware

11. IANA Considerations

11.1. WebAuthn Attestation Statement Format Identifier Registrations

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

* WebAuthn Attestation Statement Format Identifier: packed
* Description: The "packed" attestation statement format is a WebAuthn-optimized format for attestation. It uses a very compact but still extensible encoding method. This format is implementable by authenticators with limited resources (e.g., secure elements).
* Specification Document: Section 8.2 Packed Attestation Statement Format of this specification
* WebAuthn Attestation Statement Format Identifier: tpm
* Description: The TPM attestation statement format returns an attestation statement in the same format as the packed attestation statement format, although the the rawData and signature fields are computed differently.
* Specification Document: Section 8.3 TPM Attestation Statement Format of this specification
* WebAuthn Attestation Statement Format Identifier: android-key
* Description: Platform-provided authenticators based on versions "N", and later, may provide this proprietary "hardware attestation" statement.
* Specification Document: Section 8.4 Android Key Attestation Statement Format of this specification
* WebAuthn Attestation Statement Format Identifier: android-safetynet
* Description: Android-based, platform-provided authenticators may produce an attestation statement based on the Android SafetyNet API.
* Specification Document: Section 8.5 Android SafetyNet Attestation Statement Format of this specification
* WebAuthn Attestation Statement Format Identifier: fido-u2f
* Description: Used with FIDO U2F authenticators
* Specification Document: Section 8.6 FIDO U2F Attestation Statement Format of this specification

11.2. WebAuthn Extension Identifier Registrations

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

* WebAuthn Extension Identifier: appid
* Description: This authentication extension allows Relying Parties that have previously registered a credential using the legacy FIDO JavaScript APIs to request an assertion.
* Specification Document: Section 10.1 FIDO AppId Extension (appid) of this specification
* WebAuthn Extension Identifier: txAuthSimple
* Description: This registration extension and authentication extension allows for a simple form of transaction authorization. A WebAuthn Relying Party can specify a prompt string, intended for display on a trusted device on the authenticator
* Specification Document: Section 10.2 Simple Transaction Authorization Extension (txAuthSimple) of this specification
* WebAuthn Extension Identifier: txAuthGeneric
* Description: This registration extension and authentication extension allows images to be used as transaction authorization prompts as well. This allows authenticators without a font rendering engine to be used and also supports a richer visual appearance than accomplished with the webauthn.txauth.simple extension.
* Specification Document: Section 10.3 Generic Transaction Authorization Extension (txAuthGeneric) of this specification
* WebAuthn Extension Identifier: authnSel

```
3854      * Description: This registration extension allows a WebAuthn Relying
3855        Party to guide the selection of the authenticator that will be
3856        leveraged when creating the credential. It is intended primarily
3857        for WebAuthn Relying Parties that wish to tightly control the
3858        experience around credential creation.
3859      * Specification Document: Section 9.4 Authenticator Selection
3860        Extension (authnSel) of this specification
3861      * WebAuthn Extension Identifier: exts
3862      * Description: This registration extension enables the Relying Party
3863        to determine which extensions the authenticator supports. The
3864        extension data is a list (CBOR array) of extension identifiers
3865        encoded as UTF-8 Strings. This extension is added automatically by
3866        the authenticator. This extension can be added to attestation
3867        statements.
3868      * Specification Document: Section 9.5 Supported Extensions Extension
3869        (exts) of this specification
3870      * WebAuthn Extension Identifier: uvi
3871      * Description: This registration extension and authentication
3872        extension enables use of a user verification index. The user
3873        verification index is a value uniquely identifying a user
3874        verification data record. The UVI data can be used by servers to
3875        understand whether an authentication was authorized by the exact
3876        same biometric data as the initial key generation. This allows the
3877        detection and prevention of "friendly fraud".
3878      * Specification Document: Section 9.6 User Verification Index
3879        Extension (uvi) of this specification
3880      * WebAuthn Extension Identifier: loc
3881      * Description: The location registration extension and authentication
3882        extension provides the client device's current location to the
3883        WebAuthn relying party, if supported by the client device and
3884        subject to user consent.
3885      * Specification Document: Section 9.7 Location Extension (loc) of
3886        this specification
3887      * WebAuthn Extension Identifier: uvm
3888      * Description: This registration extension and authentication
3889        extension enables use of a user verification method. The user
3890        verification method extension returns to the Webauthn relying party
3891        which user verification methods (factors) were used for the
3892        WebAuthn operation.
3893      * Specification Document: Section 9.8 User Verification Method
3894        Extension (uvm) of this specification
3895
3896    10.3. COSE Algorithm Registrations
3897
3898    This section registers identifiers for RSASSA-PKCS1-v1_5 [RFC8017]
3899    algorithms using SHA-2 hash functions in the IANA COSE Algorithms
3900    registry [IANA-COSE-ALGS-REG].

3901      * Name: RS256
3902      * Value: -257
3903      * Description: RSASSA-PKCS1-v1_5 w/ SHA-256
3904      * Reference: Section 8.2 of [RFC8017]
3905      * Recommended: No
3906      * Name: RS384
3907      * Value: -258
3908      * Description: RSASSA-PKCS1-v1_5 w/ SHA-384
3909      * Reference: Section 8.2 of [RFC8017]
3910      * Recommended: No
3911      * Name: RS512
3912      * Value: -259
3913      * Description: RSASSA-PKCS1-v1_5 w/ SHA-512
3914      * Reference: Section 8.2 of [RFC8017]
3915      * Recommended: No
```

```
4639      * Description: This registration extension allows a WebAuthn Relying
4640        Party to guide the selection of the authenticator that will be
4641        leveraged when creating the credential. It is intended primarily
4642        for WebAuthn Relying Parties that wish to tightly control the
4643        experience around credential creation.
4644      * Specification Document: Section 10.4 Authenticator Selection
4645        Extension (authnSel) of this specification
4646      * WebAuthn Extension Identifier: exts
4647      * Description: This registration extension enables the Relying Party
4648        to determine which extensions the authenticator supports. The
4649        extension data is a list (CBOR array) of extension identifiers
4650        encoded as UTF-8 Strings. This extension is added automatically by
4651        the authenticator. This extension can be added to attestation
4652        statements.
4653      * Specification Document: Section 10.5 Supported Extensions
4654        Extension (exts) of this specification
4655      * WebAuthn Extension Identifier: uvi
4656      * Description: This registration extension and authentication
4657        extension enables use of a user verification index. The user
4658        verification index is a value uniquely identifying a user
4659        verification data record. The UVI data can be used by servers to
4660        understand whether an authentication was authorized by the exact
4661        same biometric data as the initial key generation. This allows the
4662        detection and prevention of "friendly fraud".
4663      * Specification Document: Section 10.6 User Verification Index
4664        Extension (uvi) of this specification
4665      * WebAuthn Extension Identifier: loc
4666      * Description: The location registration extension and authentication
4667        extension provides the client device's current location to the
4668        WebAuthn relying party, if supported by the client device and
4669        subject to user consent.
4670      * Specification Document: Section 10.7 Location Extension (loc) of
4671        this specification
4672      * WebAuthn Extension Identifier: uvm
4673      * Description: This registration extension and authentication
4674        extension enables use of a user verification method. The user
4675        verification method extension returns to the Webauthn relying party
4676        which user verification methods (factors) were used for the
4677        WebAuthn operation.
4678      * Specification Document: Section 10.8 User Verification Method
4679        Extension (uvm) of this specification
4680
4681    11.3. COSE Algorithm Registrations
4682
4683    This section registers identifiers for RSASSA-PKCS1-v1_5 [RFC8017]
4684    algorithms using SHA-2 and SHA-1 hash functions in the IANA COSE
4685    Algorithms registry [IANA-COSE-ALGS-REG]. It also registers identifiers
4686    for ECDAA algorithms.
4687      * Name: RS256
4688      * Value: -257
4689      * Description: RSASSA-PKCS1-v1_5 w/ SHA-256
4690      * Reference: Section 8.2 of [RFC8017]
4691      * Recommended: No
4692      * Name: RS384
4693      * Value: -258
4694      * Description: RSASSA-PKCS1-v1_5 w/ SHA-384
4695      * Reference: Section 8.2 of [RFC8017]
4696      * Recommended: No
4697      * Name: RS512
4698      * Value: -259
4699      * Description: RSASSA-PKCS1-v1_5 w/ SHA-512
4700      * Reference: Section 8.2 of [RFC8017]
4701      * Recommended: No
4702      * Name: ED256
4703      * Value: -260
4704      * Description: TPM_ECC_BN_P256 curve w/ SHA-256
4705      * Reference: Section 4.2 of [FIDOEcdaaAlgorithm]
4706      * Recommended: Yes
4707      * Name: ED512
4708      * Value: -261
```

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. In this flow, the Relying Party does not have a preference for platform authenticator or roaming authenticators.

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

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

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

var publicKey = {
```



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

12. Sample scenarios

This section is not normative.

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

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

12.1. Registration

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

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

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

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

var publicKey = {
```

```
    challenge: Uint8Array.from(window.atob("PGifxAoBwCkWkm4b1Cill5otCphilh6MijdjbW
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.
    pubKeyCredParams: [
      {
        type: "public-key",
        alg: -7 // "ES256" as registered in the IANA COSE Algorithms registry
      },
      {
        type: "public-key",
        alg: -257 // Value registered by this specification for "RS256"
      }
    ],

    timeout: 60000,  // 1 minute
    excludeCredentials: [], // No exclude list of PKCredDescriptors
    extensions: {"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. Registration Specifically with Platform Authenticator

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

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

    PublicKeyCredential.isPlatformAuthenticatorAvailable()
      .then(function (userIntent) {
```



```
    // The challenge must be produced by the server, see the Security Consideratio
  ns
    challenge: new Uint8Array([21,31,105 /* 29 more random bytes generated by the
  server */]),

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

    // User:
    user: {
      id: Uint8Array.from(window.atob("MIIBkzCCATigAwIBAjCCAZMwggE4oAMCAQIwggGTMII
  ="), c=>c.charCodeAt(0)),
      name: "john.p.smith@example.com",
      displayName: "John P. Smith",
      icon: "https://pics.acme.com/00/p/aBjjjpqPb.png"
    },

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

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

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

    12.2. Registration Specifically with User Verifying Platform Authenticator

    This is flow for when the Relying Party is specifically interested in
    creating a public key credential with a user-verifying platform
    authenticator.
      1. The user visits example.com and clicks on the login button, which
         redirects the user to login.example.com.
      2. The user enters a username and password to log in. After successful
         login, the user is redirected back to example.com.
      3. The Relying Party script runs the code snippet below.
      4. The user agent asks the user whether they are willing to register
         with the Relying Party using an available platform authenticator.
      5. If the user is not willing, terminate this flow.
      6. The user is shown appropriate UI and guided in creating a
         credential using one of the available platform authenticators. Upon
         successful credential creation, the RP script conveys the new
         credential to the server.
  if (!PublicKeyCredential) { /* Platform not capable of the API. Handle error. */
    }

    PublicKeyCredential.isUserVerifyingPlatformAuthenticatorAvailable()
      .then(function (userIntent) {
```

```
        // If the user has affirmed willingness to register with RP using an ava
ilable platform authenticator
        if (userIntent) {
            var publicKeyOptions = { /* Public key credential creation options.
*/};

            // Create and register credentials.
            return navigator.credentials.create({ "publicKey": publicKeyOptions
});
        } else {

            // Record that the user does not intend to use a platform authentica
tor
            // and default the user to a password-based flow in the future.
        }

    }).then(function (newCredentialInfo) {
        // Send new credential info to server for verification and registration.
    }).catch( function(err) {
        // Something went wrong. Handle appropriately.
    });
```

11.3. Authentication

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

1. The user visits example.com, which serves up a script.
2. The script asks the client platform for an Authentication Assertion, providing as much information as possible to narrow the choice of acceptable credentials for the user. This may be obtained from the data that was stored locally after registration, or by other means such as prompting the user for a username.
3. The Relying Party script runs one of the code snippets below.
4. The client platform searches for and locates the authenticator.
5. The client platform connects to the authenticator, performing any pairing actions if necessary.
6. The authenticator presents the user with a notification that their attention is required. On opening the notification, the user is shown a friendly selection menu of acceptable credentials using the 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"),
```



```
        // If the user has affirmed willingness to register with RP using an ava
ilable platform authenticator
        if (userIntent) {
            var publicKeyOptions = { /* Public key credential creation options.
*/};

            // Create and register credentials.
            return navigator.credentials.create({ "publicKey": publicKeyOptions
});
        } else {

            // Record that the user does not intend to use a platform authentica
tor
            // and default the user to a password-based flow in the future.
        }

    }).then(function (newCredentialInfo) {
        // Send new credential info to server for verification and registration.
    }).catch( function(err) {
        // Something went wrong. Handle appropriately.
    });
```

12.3. Authentication

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

1. The user visits example.com, which serves up a script.
2. The script asks the client platform for an Authentication Assertion, providing as much information as possible to narrow the choice of acceptable credentials for the user. This may be obtained from the data that was stored locally after registration, or by other means such as prompting the user for a username.
3. The Relying Party script runs one of the code snippets below.
4. The client platform searches for and locates the authenticator.
5. The client platform connects to the authenticator, performing any pairing actions if necessary.
6. The authenticator presents the user with a notification that their attention is required. On opening the notification, the user is shown a friendly selection menu of acceptable credentials using the 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 = {
        // The challenge must be produced by the server, see the Securit
```

```
                timeout: 60000,  // 1 minute
                allowCredentials: [{ type: "public-key" }]
              };

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

On the other hand, if the Relying Party script has some hints to help it narrow the list of credentials, then the sample code for performing such an authentication might look like the following. Note that this sample also demonstrates how to use the extension for transaction authorization.

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

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

var options = {
                challenge: encoder.encode("climb a mountain"),



                timeout: 60000,  // 1 minute
                allowCredentials: [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.
});
```

11.4. Decommissioning


```
y Considerations
                challenge: new Uint8Array([4,101,15 /* 29 more random bytes gene
rated by the server */]),
                timeout: 60000,  // 1 minute
                allowCredentials: [{ type: "public-key" }]
              };

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

On the other hand, if the Relying Party script has some hints to help it narrow the list of credentials, then the sample code for performing such an authentication might look like the following. Note that this sample also demonstrates how to use the extension for transaction authorization.

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

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

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

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

12.4. Aborting Authentication Operations

The below example shows how a developer may use the AbortSignal parameter to abort a credential registration operation. A similiar procedure applies to an authentication operation.

```
const authAbortController = new AbortController();
const authAbortSignal = authAbortController.signal;

authAbortSignal.onabort = function () {
    // Once the page knows the abort started, inform user it is attempting to ab
ort.
}

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

navigator.credentials.create({
    publicKey: options,
    signal: authAbortSignal})
```

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: 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.


```
    .then(function (attestation) {
        // Register the user.
    }).catch(function (error) {
        if (error == "AbortError") {
            // Inform user the credential hasn't been created.
            // Let the server know a key hasn't been created.
        }
    });

// Assume widget shows up whenever auth occurs.
if (widget == "disappear") {
    authAbortSignal.abort();

}
```

12.5. Decommissioning

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

* Possibility #1 -- user reports the credential as lost.
  + User goes to server.example.net, authenticates and follows a link to report a lost/stolen device.
  + Server returns a page showing the list of registered credentials with friendly names as configured during registration.
  + User selects a credential and the server deletes it from its database.
  + In future, the Relying Party script does not specify this credential in any list of acceptable credentials, and assertions signed by this credential are rejected.
* Possibility #2 -- server deregisters the credential due to inactivity.
  + Server deletes credential from its database during maintenance activity.
  + In the future, the Relying Party script does not specify this credential in any list of acceptable credentials, and assertions signed by this credential are rejected.
* Possibility #3 -- user deletes the credential from the device.
  + User employs a device-specific method (e.g., device settings UI) to delete a credential from their device.
  + From this point on, this credential will not appear in any selection prompts, and no assertions can be generated with it.
  + Sometime later, the server deregisters this credential due to inactivity.

13. Security Considerations

13.1. Cryptographic Challenges

As a cryptographic protocol, Web Authentication is dependent upon randomized challenges to avoid replay attacks. Therefore, both {MakePublicKeyCredentialOptions/challenge}}'s and challenge's value, MUST be randomly generated by the Relying Party in an environment they trust (e.g., on the server-side), and the challenge in the client's response must match what was generated. This should be done in a fashion that does not rely upon a client's behavior; e.g.: the Relying Party should store the challenge temporarily until the operation is complete. Tolerating a mismatch will compromise the security of the protocol.

14. Acknowledgements

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

Index

Terms defined by this specification

* aa, in 4.4.3
* AAGUID, in 9.4
* alg, in 4.3
* allowCredentials, in 4.5
* Assertion, in 3
* assertion signature, in 5
* attachment modality, in 4.4.4
* Attestation, in 3
* Attestation Certificate, in 3
* Attestation data, in 5.3.1
* attestation key pair, in 3
* attestationObject, in 4.2.1
* attestation object, in 5.3
* attestation private key, in 3
* attestation public key, in 3
* attestation signature, in 5
* attestation statement, in 5.3
* attestation statement format, in 5.3
* attestation statement format identifier, in 7.1
* attestation type, in 5.3
* Authentication, in 3
* Authentication Assertion, in 3
* authentication extension, in 8
* AuthenticationExtensions
  + definition of, in 4.6
  + (typedef), in 4.6
* Authenticator, in 3
* AuthenticatorAssertionResponse, in 4.2.2
* AuthenticatorAttachment, in 4.4.4
* AuthenticatorAttestationResponse, in 4.2.1
* authenticatorCancel, in 5.2.3
* authenticator data, in 5.1
* authenticatorData, in 4.2.2
* authenticator data claimed to have been used for the attestation, in 5.3.2
* authenticator data for the attestation, in 5.3.2
* authenticator extension, in 8
* authenticator extension input, in 8.3
* authenticator extension output, in 8.5
* Authenticator extension processing, in 8.5
* authenticatorExtensions, in 4.7.1
* authenticatorGetAssertion, in 5.2.2
* authenticatorMakeCredential, in 5.2.1
* AuthenticatorResponse, in 4.2
* authenticatorSelection, in 4.4
* AuthenticatorSelectionCriteria, in 4.4.3
* AuthenticatorSelectionList, in 9.4
* AuthenticatorTransport, in 4.7.4
* Authorization Gesture, in 3
* Base64url Encoding, in 2.1
* Basic Attestation, in 5.3.3
* Biometric Recognition, in 3
* ble, in 4.7.4
* CBOR, in 2.1
* Ceremony, in 3

Index

Terms defined by this specification

* aaguid, in 6.3.1
* AAGUID, in 10.4
* alg, in 5.3
* allowCredentials, in 5.5
* Assertion, in 4
* assertion signature, in 6
* attachment modality, in 5.4.5
* Attestation, in 4
* attestation, in 5.4
* Attestation Certificate, in 4
* Attestation Conveyance, in 5.4.6
* AttestationConveyancePreference, in 5.4.6
* attestationConveyancePreferenceOption, in 5.1.3
* attestation key pair, in 4
* attestationObject, in 5.2.1
* attestation object, in 6.3
* attestationObjectResult, in 5.1.3
* attestation private key, in 4
* attestation public key, in 4
* attestation signature, in 6
* attestation statement, in 6.3
* attestation statement format, in 6.3
* attestation statement format identifier, in 8.1
* attestation trust path, in 6.3.2
* attestation type, in 6.3
* Attested credential data, in 6.3.1
* attestedCredentialData, in 6.1
* authDataExtensions, in 6.1
* Authentication, in 4
* Authentication Assertion, in 4
* authentication extension, in 9
* AuthenticationExtensions
  + definition of, in 5.7
  + (typedef), in 5.7
* Authenticator, in 4
* AuthenticatorAssertionResponse, in 5.2.2
* AuthenticatorAttachment, in 5.4.5
* authenticatorAttachment, in 5.4.4
* AuthenticatorAttestationResponse, in 5.2.1
* authenticatorCancel, in 6.2.3
* authenticator data, in 6.1
* authenticatorData, in 5.2.2
* authenticator data claimed to have been used for the attestation, in 6.3.2
* authenticator data for the attestation, in 6.3.2
* authenticatorDataResult, in 5.1.4.1
* authenticator extension, in 9
* authenticator extension input, in 9.3
* authenticator extension output, in 9.5
* Authenticator extension processing, in 9.5
* authenticatorExtensions, in 5.8.1
* authenticatorGetAssertion, in 6.2.2
* authenticatorMakeCredential, in 6.2.1
* AuthenticatorResponse, in 5.2
* authenticatorSelection, in 5.4
* AuthenticatorSelectionCriteria, in 5.4.4
* AuthenticatorSelectionList, in 10.4
* authenticator session, in 6.2
* AuthenticatorTransport, in 5.8.4
* Authorization Gesture, in 4
* Base64url Encoding, in 3
* Basic Attestation, in 6.3.3
* Biometric Recognition, in 4
* ble, in 5.8.4
* CBOR, in 3
* Ceremony, in 4

```
4257        * challenge
4258            + dict-member for MakePublicKeyCredentialOptions, in 4.4
4259            + dict-member for PublicKeyCredentialRequestOptions, in 4.5
4260            + dict-member for CollectedClientData, in 4.7.1
4261        * Client, in 3
4262        * client data, in 4.7.1
4263        * clientDataJSON, in 4.2
4264        * client extension, in 8
4265        * client extension input, in 8.3
4266        * client extension output, in 8.4
4267        * Client extension processing, in 8.4
4268        * clientExtensionResults, in 4.1
4269        * clientExtensions, in 4.7.1
4270        * Client-Side, in 3
4271        * client-side credential private key storage, in 3
4272        * Client-side-resident Credential Private Key, in 3
4273        * CollectedClientData, in 4.7.1
4274        * Conforming User Agent, in 3
4275        * COSEAlgorithmIdentifier
4276            + definition of, in 4.7.5
4277            + (typedef), in 4.7.5
4278        * [[Create]](options), in 4.1.3
4279        * credential key pair, in 3
4280        * credential private key, in 3
4281        * Credential Public Key, in 3
4282        * cross-platform attached, in 4.4.4
4283        * cross-platform attachment, in 4.4.4
4284        * DAA, in 5.3.3
4285        * [[DiscoverFromExternalSource]](options), in 4.1.4
4286        * [[discovery]], in 4.1
4287        * displayName, in 4.4.2
4288        * ECDAA, in 5.3.3
4289        * ECDAA-Issuer public key, in 7.2
4290        * Elliptic Curve based Direct Anonymous Attestation, in 5.3.3
4291        * excludeCredentials, in 4.4
4292        * extension identifier, in 8.1
4293        * extensions
4294            + dict-member for MakePublicKeyCredentialOptions, in 4.4
4295            + dict-member for PublicKeyCredentialRequestOptions, in 4.5
4296        * hashAlgorithm, in 4.7.1
4297        * Hash of the serialized client data, in 4.7.1
4298        * icon, in 4.4.1
4299        * id
4300            + dict-member for PublicKeyCredentialEntity, in 4.4.1
4301            + dict-member for PublicKeyCredentialDescriptor, in 4.7.3
4302        * [[identifier]], in 4.1
```

```
5129        * challenge
5130            + dict-member for MakePublicKeyCredentialOptions, in 5.4
5131            + dict-member for PublicKeyCredentialRequestOptions, in 5.5
5132            + dict-member for CollectedClientData, in 5.8.1
5133        * Client, in 4
5134        * client data, in 5.8.1
5135        * clientDataJSON, in 5.2
5136        * clientDataJSONResult
5137            + dfn for credentialCreationData, in 5.1.3
5138            + dfn for assertionCreationData, in 5.1.4.1
5139        * client extension, in 9
5140        * client extension input, in 9.3
5141        * client extension output, in 9.4
5142        * Client extension processing, in 9.4
5143        * clientExtensionResults
5144            + dfn for credentialCreationData, in 5.1.3
5145            + dfn for assertionCreationData, in 5.1.4.1
5146        * clientExtensions, in 5.8.1
5147        * [[clientExtensionsResults]], in 5.1
5148        * Client-Side, in 4
5149        * client-side credential private key storage, in 4
5150        * Client-side-resident Credential Private Key, in 4
5151        * CollectedClientData, in 5.8.1
5152        * [[CollectFromCredentialStore]](origin, options,
5153          sameOriginWithAncestors), in 5.1.4
5154        * Conforming User Agent, in 4
5155        * COSEAlgorithmIdentifier
5156            + definition of, in 5.8.5
5157            + (typedef), in 5.8.5
5158        * [[Create]](origin, options, sameOriginWithAncestors), in 5.1.3
5159        * Credential ID, in 4
5160        * credentialId, in 6.3.1
5161        * credentialIdLength, in 6.3.1
5162        * credentialIdResult, in 5.1.4.1
5163        * credential key pair, in 4
5164        * credential private key, in 4
5165        * Credential Public Key, in 4
5166        * credentialPublicKey, in 6.3.1
5167        * "cross-platform", in 5.4.5
5168        * cross-platform, in 5.4.5
5169        * cross-platform attached, in 5.4.5
5170        * cross-platform attachment, in 5.4.5
5171        * DAA, in 6.3.3
5172        * direct, in 5.4.6
5173        * "discouraged", in 5.8.6
5174        * discouraged, in 5.8.6
5175        * [[DiscoverFromExternalSource]](origin, options,
5176          sameOriginWithAncestors), in 5.1.4.1
5177        * [[discovery]], in 5.1
5178        * displayName, in 5.4.3
5179        * ECDAA, in 6.3.3
5180        * ECDAA-Issuer public key, in 8.2
5181        * effective user verification requirement for assertion, in 5.1.4.1
5182        * effective user verification requirement for credential creation, in
5183          5.1.3
5184        * Elliptic Curve based Direct Anonymous Attestation, in 6.3.3
5185        * excludeCredentials, in 5.4
5186        * extension identifier, in 9.1
5187        * extensions
5188            + dict-member for MakePublicKeyCredentialOptions, in 5.4
5189            + dict-member for PublicKeyCredentialRequestOptions, in 5.5
5190        * flags, in 6.1
5191        * getClientExtensionResults(), in 5.1
5192        * hashAlgorithm, in 5.8.1
5193        * Hash of the serialized client data, in 5.8.1
5194        * icon, in 5.4.1
5195        * id
5196            + dict-member for PublicKeyCredentialRpEntity, in 5.4.2
5197            + dict-member for PublicKeyCredentialUserEntity, in 5.4.3
5198            + dict-member for PublicKeyCredentialDescriptor, in 5.8.3
```

```
4303        * identifier of the ECDAA-Issuer public key, in 7.2
4304        * isPlatformAuthenticatorAvailable(), in 4.1.5
4305        * JSON-serialized client data, in 4.7.1
4306        * MakePublicKeyCredentialOptions, in 4.4
4307        * name, in 4.4.1
4308        * nfc, in 4.7.4
4309        * origin, in 4.7.1
4310        * "plat", in 4.4.4
4311        * plat, in 4.4.4
4312        * platform attachment, in 4.4.4
4313        * platform authenticators, in 4.4.4
4314        * Privacy CA, in 5.3.3
4315        * pubKeyCredParams, in 4.4
4316        * publicKey
4317            + dict-member for CredentialCreationOptions, in 4.1.1
4318            + dict-member for CredentialRequestOptions, in 4.1.2
4319        * public-key, in 4.7.2
4320        * Public Key Credential, in 3
4321        * PublicKeyCredential, in 4.1
4322        * PublicKeyCredentialDescriptor, in 4.7.3
4323        * PublicKeyCredentialEntity, in 4.4.1
4324        * PublicKeyCredentialParameters, in 4.3
4325        * PublicKeyCredentialRequestOptions, in 4.5
4326        * PublicKeyCredentialType, in 4.7.2
4327        * PublicKeyCredentialUserEntity, in 4.4.2
4328        * Rate Limiting, in 3
4329        * rawId, in 4.1
4330        * Registration, in 3
4331        * registration extension, in 8
4332        * Relying Party, in 3
4333        * Relying Party Identifier, in 3
4334        * response, in 4.1
4335        * rk, in 4.4.3
4336        * roaming authenticators, in 4.4.4
4337        * rp, in 4.4
4338        * rpId, in 4.5
4339        * RP ID, in 3
4340        * Self Attestation, in 5.3.3
4341        * signature, in 4.2.2
4342        * Signing procedure, in 5.3.2
4343        * Test of User Presence, in 3
4344        * timeout
4345            + dict-member for MakePublicKeyCredentialOptions, in 4.4
4346            + dict-member for PublicKeyCredentialRequestOptions, in 4.5
4347        * tokenBindingId, in 4.7.1
4348        * transports, in 4.7.3
4349        * [[type]], in 4.1
4350        * type
4351            + dict-member for PublicKeyCredentialParameters, in 4.3
4352            + dict-member for PublicKeyCredentialDescriptor, in 4.7.3
4353        * UP, in 3
4354        * usb, in 4.7.4
4355        * user, in 4.4
4356        * User Consent, in 3
4357        * User Present, in 3
```

```
5199        * [[identifier]], in 5.1
5200        * identifier of the ECDAA-Issuer public key, in 8.2
5201        * indirect, in 5.4.6
5202        * isUserVerifyingPlatformAuthenticatorAvailable(), in 5.1.6
5203        * JSON-serialized client data, in 5.8.1
5204        * MakePublicKeyCredentialOptions, in 5.4
5205        * managing authenticator, in 4
5206        * name, in 5.4.1
5207        * nfc, in 5.8.4
5208        * none, in 5.4.6
5209        * origin, in 5.8.1
5210        * platform, in 5.4.5
5211        * "platform", in 5.4.5
5212        * platform attachment, in 5.4.5
5213        * platform authenticators, in 5.4.5
5214        * "preferred", in 5.8.6
5215        * preferred, in 5.8.6
5216        * Privacy CA, in 6.3.3
5217        * pubKeyCredParams, in 5.4
5218        * publicKey
5219            + dict-member for CredentialCreationOptions, in 5.1.1
5220            + dict-member for CredentialRequestOptions, in 5.1.2
5221        * public-key, in 5.8.2
5222        * Public Key Credential, in 4
5223        * PublicKeyCredential, in 5.1
5224        * PublicKeyCredentialDescriptor, in 5.8.3
5225        * PublicKeyCredentialEntity, in 5.4.1
5226        * PublicKeyCredentialParameters, in 5.3
5227        * PublicKeyCredentialRequestOptions, in 5.5
5228        * PublicKeyCredentialRpEntity, in 5.4.2
5229        * Public Key Credential Source, in 4
5230        * PublicKeyCredentialType, in 5.8.2
5231        * PublicKeyCredentialUserEntity, in 5.4.3
5232        * Rate Limiting, in 4
5233        * rawId, in 5.1
5234        * Registration, in 4
5235        * registration extension, in 9
5236        * Relying Party, in 4
5237        * Relying Party Identifier, in 4
5238        * "required", in 5.8.6
5239        * required, in 5.8.6
5240        * requireResidentKey, in 5.4.4
5241        * response, in 5.1
5242        * roaming authenticators, in 5.4.5
5243        * rp, in 5.4
5244        * rpId, in 5.5
5245        * RP ID, in 4
5246        * rpIdHash, in 6.1
5247        * Self Attestation, in 6.3.3
5248        * signature, in 5.2.2
5249        * Signature Counter, in 6.1.1
5250        * signatureResult, in 5.1.4.1
5251        * signCount, in 6.1
5252        * Signing procedure, in 6.3.2
5253        * [[Store]](credential, sameOriginWithAncestors), in 5.1.5
5254        * Test of User Presence, in 4
5255        * timeout
5256            + dict-member for MakePublicKeyCredentialOptions, in 5.4
5257            + dict-member for PublicKeyCredentialRequestOptions, in 5.5
5258        * tokenBindingId, in 5.8.1
5259        * transports, in 5.8.3
5260        * [[type]], in 5.1
5261        * type
5262            + dict-member for PublicKeyCredentialParameters, in 5.3
5263            + dict-member for CollectedClientData, in 5.8.1
5264            + dict-member for PublicKeyCredentialDescriptor, in 5.8.3
5265        * UP, in 4
5266        * usb, in 5.8.4
5267        * user, in 5.4
5268        * User Consent, in 4
```

```
4358        * User Verification, in 3                          5269        * userHandle, in 5.2.2
4359        * User Verified, in 3                              5270        * User Handle, in 4
4360        * UV, in 3                                         5271        * userHandleResult, in 5.1.4.1
4361        * uv, in 4.4.3                                     5272        * User Present, in 4
4362        * Verification procedures, in 5.3.2                5273        * userVerification
4363        * Web Authentication API, in 4                     5274          + dict-member for AuthenticatorSelectionCriteria, in 5.4.4
4364        * WebAuthn Client, in 3                            5275          + dict-member for PublicKeyCredentialRequestOptions, in 5.5
4365        * "xplat", in 4.4.4                                5276        * User Verification, in 4
4366        * xplat, in 4.4.4                                  5277        * UserVerificationRequirement, in 5.8.6
                                                               5278        * User Verified, in 4
                                                               5279        * UV, in 4
                                                               5280        * Verification procedure, in 6.3.2
                                                               5281        * verification procedure inputs, in 6.3.2
                                                               5282        * Web Authentication API, in 5
                                                               5283        * WebAuthn Client, in 4
4367                                                           5284
4368    Terms defined by reference                             5285    Terms defined by reference
4369                                                           5286
4370      * [CREDENTIAL-MANAGEMENT-1] defines the following terms:  5287      * [CREDENTIAL-MANAGEMENT-1] defines the following terms:
4371          + Credential                                     5288          + Credential
4372          + CredentialCreationOptions                      5289          + CredentialCreationOptions
4373          + CredentialRequestOptions                       5290          + CredentialRequestOptions
4374          + CredentialsContainer                           5291          + CredentialsContainer
4375          + [[CollectFromCredentialStore]](options)        5292          + Request a Credential
4376          + [[Store]](credential)                          5293          + [[CollectFromCredentialStore]](origin, options,
                                                               5294            sameOriginWithAncestors)
                                                               5295          + [[Create]](origin, options, sameOriginWithAncestors)
                                                               5296          + [[Store]](credential, sameOriginWithAncestors)
4377          + [[discovery]]                                  5297          + [[discovery]]
4378          + [[type]]                                       5298          + [[type]]
4379          + create()                                       5299          + create()
                                                               5300          + credential
                                                               5301          + credential source
4380          + get()                                          5302          + get()
4381          + id                                             5303          + id
4382          + remote                                         5304          + remote
                                                               5305          + same-origin with its ancestors
                                                               5306          + signal (for CredentialCreationOptions)
                                                               5307          + signal (for CredentialRequestOptions)
                                                               5308          + store()
4383          + type                                           5309          + type
                                                               5310          + user mediation
                                                               5311        * [DOM4] defines the following terms:
                                                               5312          + AbortController
                                                               5313          + aborted flag
                                                               5314          + document
4384      * [ECMAScript] defines the following terms:          5315      * [ECMAScript] defines the following terms:
4385          + %arraybuffer%                                  5316          + %arraybuffer%
                                                               5317          + internal method
4386          + internal slot                                  5318          + internal slot
4387          + stringify                                      5319          + stringify
4388      * [ENCODING] defines the following terms:           5320      * [ENCODING] defines the following terms:
4389          + utf-8 encode                                   5321          + utf-8 encode
                                                               5322        * [FETCH] defines the following terms:
                                                               5323          + window
4390      * [HTML] defines the following terms:               5324      * [HTML] defines the following terms:
4391          + ascii serialization of an origin               5325          + ascii serialization of an origin
4392          + dom manipulation task source
4393          + effective domain                               5326          + effective domain
                                                               5327          + environment settings object
4394          + global object                                  5328          + global object
4395          + in parallel
4396          + is a registrable domain suffix of or is equal to  5329          + is a registrable domain suffix of or is equal to
4397          + is not a registrable domain suffix of and is not equal to  5330          + is not a registrable domain suffix of and is not equal to
4398          + origin                                         5331          + origin
4399          + promise
4400          + relevant settings object                       5332          + relevant settings object
4401          + task
4402          + task source
4403      * [HTML52] defines the following terms:             5333      * [HTML52] defines the following terms:
```

```
4404          + document.domain
4405          + opaque origin
4406          + origin
4407       * [INFRA] defines the following terms:
4408          + append (for list)
4409          + append (for set)

4410          + continue

4411          + for each (for list)
4412          + for each (for map)
4413          + is empty
4414          + is not empty
4415          + item

4416          + list
4417          + map
4418          + ordered set
4419          + remove
4420          + set

4421       * [secure-contexts] defines the following terms:
4422          + secure context
4423       * [TokenBinding] defines the following terms:
4424          + token binding
4425          + token binding id
4426       * [URL] defines the following terms:
4427          + domain
4428          + empty host
4429          + host
4430          + ipv4 address
4431          + ipv6 address
4432          + opaque host
4433          + url serializer
4434          + valid domain
4435          + valid domain string
4436       * [WebCryptoAPI] defines the following terms:
4437          + recognized algorithm name
4438       * [WebIDL] defines the following terms:

4439          + ArrayBuffer
4440          + BufferSource
4441          + ConstraintError
4442          + DOMException
4443          + DOMString

4444          + NotAllowedError
4445          + NotFoundError
4446          + NotSupportedError
4447          + Promise
4448          + SameObject
4449          + SecureContext
4450          + SecurityError
4451          + TypeError
4452          + USVString
4453          + UnknownError
4454          + Unscopable
4455          + boolean
4456          + interface object
4457          + long
4458          + present
4459          + simple exception
4460          + unsigned long
```

```
5334          + document.domain
5335          + opaque origin
5336          + origin
5337       * [INFRA] defines the following terms:
5338          + append (for list)
5339          + append (for set)
5340          + byte sequence
5341          + continue
5342          + empty
5343          + for each (for list)
5344          + for each (for map)
5345          + is empty
5346          + is not empty
5347          + item (for list)
5348          + item (for struct)
5349          + list
5350          + map
5351          + ordered set
5352          + remove
5353          + set
5354          + size
5355          + struct
5356          + while
5357          + willful violation
5358       * [mixed-content] defines the following terms:
5359          + a priori authenticated url
5360       * [page-visibility] defines the following terms:
5361          + visibility states
5362       * [secure-contexts] defines the following terms:
5363          + secure contexts
5364       * [TokenBinding] defines the following terms:
5365          + token binding
5366          + token binding id
5367       * [URL] defines the following terms:
5368          + domain
5369          + empty host
5370          + host
5371          + ipv4 address
5372          + ipv6 address
5373          + opaque host
5374          + url serializer
5375          + valid domain
5376          + valid domain string
5377       * [WebCryptoAPI] defines the following terms:
5378          + recognized algorithm name
5379       * [WebIDL] defines the following terms:
5380          + AbortError
5381          + ArrayBuffer
5382          + BufferSource
5383          + ConstraintError
5384          + DOMException
5385          + DOMString
5386          + Exposed
5387          + NotAllowedError

5388          + NotSupportedError
5389          + Promise
5390          + SameObject
5391          + SecureContext
5392          + SecurityError

5393          + USVString
5394          + UnknownError

5395          + boolean
5396          + interface object
5397          + long
5398          + present

5399          + 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; 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/

[IANA-COSE-ALGS-REG]
IANA CBOR Object Signing and Encryption (COSE) Algorithms Registry. URL: https://www.iana.org/assignments/cose/cose.xhtml#algorithms

* [whatwg html] defines the following terms:
  + focus

References

Normative References

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

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

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

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

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

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

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

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

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

[FIDOReg]
R. Lindemann; D. Baghdasaryan; B. Hill. FIDO UAF Registry of Predefined Values. FIDO Alliance Proposed Standard. URL: https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-reg-v1.0-ps-20141208.html

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

[HTML52]
Steve Faulkner; et al. HTML 5.2. 2 November 2017. PR. URL: https://www.w3.org/TR/html52/

[IANA-COSE-ALGS-REG]
IANA CBOR Object Signing and Encryption (COSE) Algorithms Registry. URL: https://www.iana.org/assignments/cose/cose.xhtml#algorithms

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

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

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

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

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

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

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

[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/

[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/

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

[MIXED-CONTENT]
Mike West. Mixed Content. 2 August 2016. CR. URL: https://www.w3.org/TR/mixed-content/

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

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

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

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

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

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

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

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

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

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

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

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

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

[WebIDL]
Cameron McCormack; Boris Zbarsky; Tobie Langel. Web IDL. 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

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

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

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

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

Informative References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[TPMv2-Part1]
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 {
    [SameObject] readonly attribute ArrayBuffer              rawId;
    [SameObject] readonly attribute AuthenticatorResponse    response;
    [SameObject] readonly attribute AuthenticationExtensions clientExtensionResults;
};

partial dictionary CredentialCreationOptions {
    MakePublicKeyCredentialOptions      publicKey;
};

partial dictionary CredentialRequestOptions {
    PublicKeyCredentialRequestOptions      publicKey;
};

[SecureContext]
partial interface PublicKeyCredential {
    [Unscopable] Promise < boolean > isPlatformAuthenticatorAvailable();
};

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

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


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

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

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

[TPMv2-Part1]
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, Exposed=Window]
interface PublicKeyCredential : Credential {
    [SameObject] readonly attribute ArrayBuffer              rawId;
    [SameObject] readonly attribute AuthenticatorResponse    response;
    AuthenticationExtensions getClientExtensionResults();
};

partial dictionary CredentialCreationOptions {
    MakePublicKeyCredentialOptions      publicKey;
};

partial dictionary CredentialRequestOptions {
    PublicKeyCredentialRequestOptions      publicKey;
};

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

[SecureContext, Exposed=Window]
interface AuthenticatorResponse {
    [SameObject] readonly attribute ArrayBuffer      clientDataJSON;
};

[SecureContext, Exposed=Window]
interface AuthenticatorAttestationResponse : AuthenticatorResponse {
    [SameObject] readonly attribute ArrayBuffer      attestationObject;
};
```

/Users/jehodges/Documents/work/standards/W3C/webauthn/index-master-tr-598ac41-WD-06.txt, Top line: 4707

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

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

dictionary MakePublicKeyCredentialOptions {
    required PublicKeyCredentialEntity          rp;
    required PublicKeyCredentialUserEntity      user;

    required BufferSource                       challenge;
    required sequence<PublicKeyCredentialParameters>  pubKeyCredParams;

    unsigned long                               timeout;
    sequence<PublicKeyCredentialDescriptor>     excludeCredentials = [];
    AuthenticatorSelectionCriteria              authenticatorSelection;

    AuthenticationExtensions                    extensions;
};

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

dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
    DOMString     displayName;

};

dictionary AuthenticatorSelectionCriteria {
    AuthenticatorAttachment     aa;       // authenticatorAttachment
    boolean                     rk = false; // requireResidentKey
    boolean                     uv = false; // requireUserVerification
};

enum AuthenticatorAttachment {
    "plat",  // Platform attachment
    "xplat"  // Cross-platform attachment

};

dictionary PublicKeyCredentialRequestOptions {
    required BufferSource               challenge;
    unsigned long                       timeout;
    USVString                           rpId;
    sequence<PublicKeyCredentialDescriptor> allowCredentials = [];

    AuthenticationExtensions            extensions;
};

typedef record<DOMString, any>      AuthenticationExtensions;

dictionary CollectedClientData {
```

/Users/jehodges/Documents/work/standards/W3C/webauthn/index-master-tr-5e63e57-WD-07.txt, Top line: 5678

```
[SecureContext, Exposed=Window]
interface AuthenticatorAssertionResponse : AuthenticatorResponse {
    [SameObject] readonly attribute ArrayBuffer      authenticatorData;
    [SameObject] readonly attribute ArrayBuffer      signature;
    [SameObject] readonly attribute ArrayBuffer      userHandle;
};

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

dictionary MakePublicKeyCredentialOptions {
    required PublicKeyCredentialRpEntity        rp;
    required PublicKeyCredentialUserEntity      user;

    required BufferSource                       challenge;
    required sequence<PublicKeyCredentialParameters>  pubKeyCredParams;

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

dictionary PublicKeyCredentialEntity {
    required DOMString    name;

    USVString             icon;
};

dictionary PublicKeyCredentialRpEntity : PublicKeyCredentialEntity {
    DOMString     id;
};

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

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

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

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

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

typedef record<DOMString, any>      AuthenticationExtensions;

dictionary CollectedClientData {
```

```
    required DOMString          challenge;
    required DOMString          origin;
    required DOMString          hashAlgorithm;
    DOMString                   tokenBindingId;
    AuthenticationExtensions    clientExtensions;
    AuthenticationExtensions    authenticatorExtensions;
};

enum PublicKeyCredentialType {
    "public-key"
};

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

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

typedef long COSEAlgorithmIdentifier;









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)


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

enum PublicKeyCredentialType {
    "public-key"
};

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

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

typedef long COSEAlgorithmIdentifier;

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

typedef sequence<AAGUID>    AuthenticatorSelectionList;

typedef BufferSource    AAGUID;

Issues Index

  The definitions of "lifetime of" and "becomes available" are intended
  to represent how devices are hotplugged into (USB) or discovered by
  (NFC) browsers, and are under-specified. Resolving this with good
  definitions or some other means will be addressed by resolving Issue
  #613. RET
  need to define "blinding". See also #462.
  <https://github.com/w3c/webauthn/issues/694> RET
  @balfanz wishes to add to the "direct" case: If the authenticator
  violates the privacy requirements of the attestation type it is using,
  the client SHOULD terminate this algorithm with a
  "AttestationNotPrivateError". RET
  The definitions of "lifetime of" and "becomes available" are intended
  to represent how devices are hotplugged into (USB) or discovered by
  (NFC) browsers, and are under-specified. Resolving this with good
  definitions or some other means will be addressed by resolving Issue
  #613. RET
  The foregoing step _may_ be incorrect, in that we are attempting to
  create savedCredentialId here and use it later below, and we do not
  have a global in which to allocate a place for it. Perhaps this is good
  enough? addendum: @jcjones feels the above step is likely good enough.
  RET
  The WHATWG HTML WG is discussing whether to provide a hook when a
  browsing context gains or loses focuses. If a hook is provided, the
  above paragraph will be updated to include the hook. See WHATWG HTML WG
  Issue #2711 for more details. RET
```

#base64url-encodingReferenced in:
* 5.1. PublicKeyCredential Interface
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2)

```
4798      * 4.1.4. Use an existing credential to make an assertion -
4799        PublicKeyCredential's [[DiscoverFromExternalSource]](options)
4800        method (2)
4801      * 6.2. Verifying an authentication assertion
4802
4803    #cborReferenced in:
4804      * 4.1.3. Create a new credential - PublicKeyCredential's
4805        [[Create]](options) method
4806      * 4.1.4. Use an existing credential to make an assertion -
4807        PublicKeyCredential's [[DiscoverFromExternalSource]](options)
4808        method
4809      * 5.1. Authenticator data (2)
4810      * 8. WebAuthn Extensions (2) (3)
4811      * 8.2. Defining extensions (2)
4812      * 8.3. Extending request parameters
4813      * 8.4. Client extension processing (2)
4814      * 8.5. Authenticator extension processing (2) (3) (4) (5)
4815
4816    #attestationReferenced in:
4817      * 3. Terminology
4818      * 5. WebAuthn Authenticator model (2)
4819      * 5.3. Attestation (2) (3) (4)
4820
4821    #attestation-certificateReferenced in:
4822      * 3. Terminology (2)
4823      * 7.3.1. TPM attestation statement certificate requirements
4824
4825    #attestation-key-pairReferenced in:
4826      * 3. Terminology (2)
4827      * 5.3. Attestation
4828
4829    #attestation-private-keyReferenced in:
4830      * 5. WebAuthn Authenticator model
4831      * 5.3. Attestation
4832
4833    #attestation-public-keyReferenced in:
4834      * 5.3. Attestation
4835
4836    #authenticationReferenced in:
4837      * 1. Introduction (2)
4838      * 3. Terminology (2) (3) (4) (5) (6) (7)
4839      * 6.2. Verifying an authentication assertion
4840
4841    #authentication-assertionReferenced in:
4842      * 1. Introduction
4843      * 3. Terminology (2) (3)
4844      * 4.1. PublicKeyCredential Interface
4845      * 4.2.2. Web Authentication Assertion (interface
4846        AuthenticatorAssertionResponse)
4847      * 4.5. Options for Assertion Generation (dictionary
4848        PublicKeyCredentialRequestOptions)
4849      * 8. WebAuthn Extensions
4850
4851    #authenticatorReferenced in:
4852      * 1. Introduction (2) (3) (4)
4853      * 1.1. Use Cases
4854      * 2. Conformance
4855      * 3. Terminology (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13)
4856        (14) (15)
4857      * 4. Web Authentication API (2) (3)
4858      * 4.1. PublicKeyCredential Interface
4859      * 4.1.3. Create a new credential - PublicKeyCredential's
4860        [[Create]](options) method (2)
4861      * 4.1.4. Use an existing credential to make an assertion -
```

```
5817      * 5.1.4.1. PublicKeyCredential's
5818        [[DiscoverFromExternalSource]](origin, options,
5819        sameOriginWithAncestors) method (2)
5820      * 7.2. Verifying an authentication assertion
5821
5822    #cborReferenced in:
5823      * 5.1.3. Create a new credential - PublicKeyCredential's
5824        [[Create]](origin, options, sameOriginWithAncestors) method
5825      * 5.1.4.1. PublicKeyCredential's
5826        [[DiscoverFromExternalSource]](origin, options,
5827        sameOriginWithAncestors) method
5828      * 6.1. Authenticator data (2)
5829      * 9. WebAuthn Extensions (2) (3)
5830      * 9.2. Defining extensions (2)
5831      * 9.3. Extending request parameters
5832      * 9.4. Client extension processing (2)
5833      * 9.5. Authenticator extension processing (2) (3) (4) (5)
5834
5835    #attestationReferenced in:
5836      * 4. Terminology (2)
5837      * 5.4.6. Attestation Conveyance Preference enumeration (enum
5838        AttestationConveyancePreference) (2)
5839      * 6. WebAuthn Authenticator model (2)
5840      * 6.3. Attestation (2) (3) (4)
5841      * 11.1. WebAuthn Attestation Statement Format Identifier
5842        Registrations
5843
5844    #attestation-certificateReferenced in:
5845      * 4. Terminology (2)
5846      * 5.1.3. Create a new credential - PublicKeyCredential's
5847        [[Create]](origin, options, sameOriginWithAncestors) method
5848      * 8.3.1. TPM attestation statement certificate requirements
5849
5850    #attestation-key-pairReferenced in:
5851      * 4. Terminology (2)
5852      * 6.3. Attestation
5853
5854    #attestation-private-keyReferenced in:
5855      * 6. WebAuthn Authenticator model
5856      * 6.3. Attestation
5857
5858    #attestation-public-keyReferenced in:
5859      * 6.3. Attestation
5860
5861    #authenticationReferenced in:
5862      * 1. Introduction (2)
5863      * 4. Terminology (2) (3) (4) (5) (6) (7)
5864      * 7.2. Verifying an authentication assertion (2) (3)
5865
5866    #authentication-assertionReferenced in:
5867      * 1. Introduction
5868      * 4. Terminology (2) (3) (4) (5) (6) (7) (8)
5869      * 5.1. PublicKeyCredential Interface
5870      * 5.2.2. Web Authentication Assertion (interface
5871        AuthenticatorAssertionResponse)
5872      * 5.5. Options for Assertion Generation (dictionary
5873        PublicKeyCredentialRequestOptions)
5874      * 9. WebAuthn Extensions
5875
5876    #authenticatorReferenced in:
5877      * 1. Introduction (2) (3) (4)
5878      * 1.1. Use Cases
5879      * 2.2. Authenticators
5880      * 4. Terminology (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13)
5881        (14) (15) (16) (17)
5882      * 5. Web Authentication API (2) (3)
5883      * 5.1. PublicKeyCredential Interface
5884      * 5.1.3. Create a new credential - PublicKeyCredential's
5885        [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
5886      * 5.1.4.1. PublicKeyCredential's
```

PublicKeyCredential's [[DiscoverFromExternalSource]](options) method (2) (3)
* 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)
* 4.4.4. Authenticator Attachment enumeration (enum AuthenticatorAttachment)
* 4.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)
* 5. WebAuthn Authenticator model (2) (3) (4) (5) (6)
* 5.1. Authenticator data
* 5.2.1. The authenticatorMakeCredential operation
* 5.2.2. The authenticatorGetAssertion operation (2) (3)
* 5.3. Attestation (2) (3) (4) (5) (6) (7) (8) (9)
* 5.3.2. Attestation Statement Formats
* 5.3.4. Generating an Attestation Object (2)
* 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
* 4.1.5. Platform Authenticator Availability - PublicKeyCredential's isPlatformAuthenticatorAvailable() method (2) (3) (4)

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

#conforming-user-agentReferenced in:
* 1. Introduction

[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method (2) (3) (4) (5)
* 5.2. Authenticator Responses (interface AuthenticatorResponse)
* 5.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse) (2)
* 5.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
* 5.4.1. Public Key Entity Description (dictionary PublicKeyCredentialEntity) (2)
* 5.4.3. User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity)
* 5.4.5. Authenticator Attachment enumeration (enum AuthenticatorAttachment)
* 5.4.6. Attestation Conveyance Preference enumeration (enum AttestationConveyancePreference) (2)
* 5.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)
* 6. WebAuthn Authenticator model (2) (3) (4) (5) (6)
* 6.1. Authenticator data
* 6.2.1. The authenticatorMakeCredential operation (2) (3)
* 6.2.2. The authenticatorGetAssertion operation (2) (3)
* 6.3. Attestation (2) (3) (4) (5) (6) (7) (8) (9)
* 6.3.2. Attestation Statement Formats
* 6.3.4. Generating an Attestation Object
* 6.3.5.1. Privacy
* 6.3.5.2. Attestation Certificate and Attestation Certificate CA Compromise
* 7.1. Registering a new credential
* 8.2. Packed Attestation Statement Format
* 8.4. Android Key Attestation Statement Format
* 8.5. Android SafetyNet Attestation Statement Format
* 10.5. Supported Extensions Extension (exts)
* 10.6. User Verification Index Extension (uvi)
* 10.7. Location Extension (loc) (2) (3) (4)
* 10.8. User Verification Method Extension (uvm)
* 12. Sample scenarios

#authorization-gestureReferenced in:
* 1.1.1. Registration
* 1.1.2. Authentication
* 1.1.3. Other use cases and configurations
* 4. Terminology (2) (3) (4) (5) (6)
* 5.1.4. Use an existing credential to make an assertion - PublicKeyCredential's [[Get]](options) method (2)

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

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

#clientReferenced in:
* 4. Terminology
* 5.1.6. Availability of User-Verifying Platform Authenticator - PublicKeyCredential's isUserVerifyingPlatformAuthenticatorAvailable() method (2) (3) (4)

#client-side-resident-credential-private-keyReferenced in:
* 4. Terminology (2)
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
* 5.4.4. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria) (2)
* 6.2.1. The authenticatorMakeCredential operation

#conforming-user-agentReferenced in:
* 1. Introduction

```
4923        * 2. Conformance (2) (3)
4924        * 3. Terminology (2)
4925
4926    #credential-public-keyReferenced in:
4927      * 3. Terminology (2) (3)
4928      * 4.2.1. Information about Public Key Credential (interface
4929        AuthenticatorAttestationResponse)
4930      * 5. WebAuthn Authenticator model
4931      * 5.1. Authenticator data
4932      * 5.3. Attestation (2) (3)
4933      * 5.3.1. Attestation data (2)
4934      * 7.4. Android Key Attestation Statement Format
4935      * 11.1. Registration (2)
4936
4937    #credential-key-pairReferenced in:
4938      * 3. Terminology (2) (3)
4939      * 4.1.3. Create a new credential - PublicKeyCredential's
4940        [[Create]](options) method
4941
4942    #credential-private-keyReferenced in:
4943      * 3. Terminology (2) (3) (4)
4944      * 4.1. PublicKeyCredential Interface
4945      * 4.2.2. Web Authentication Assertion (interface
4946        AuthenticatorAssertionResponse)
4947      * 5. WebAuthn Authenticator model
4948      * 5.2.2. The authenticatorGetAssertion operation
4949      * 5.3. Attestation (2)
```

```
5957        * 2.1. User Agents
5958        * 2.2. Authenticators
5959        * 4. Terminology (2)
5960
5961      #credential-idReferenced in:
5962        * 4. Terminology (2) (3) (4)
5963        * 5.1.4.1. PublicKeyCredential's
5964          [[DiscoverFromExternalSource]](origin, options,
5965          sameOriginWithAncestors) method
5966        * 5.2.1. Information about Public Key Credential (interface
5967          AuthenticatorAttestationResponse)
5968        * 6.2.2. The authenticatorGetAssertion operation (2)
5969        * 6.3.1. Attested credential data
5970        * 7.1. Registering a new credential
5971        * 8.6. FIDO U2F Attestation Statement Format
5972        * 12.1. Registration
5973        * 12.3. Authentication (2) (3)
5974
5975      #credential-public-keyReferenced in:
5976        * 4. Terminology (2) (3) (4) (5) (6) (7)
5977        * 5.2.1. Information about Public Key Credential (interface
5978          AuthenticatorAttestationResponse)
5979        * 6. WebAuthn Authenticator model
5980        * 6.3. Attestation (2) (3)
5981        * 6.3.1. Attested credential data (2)
5982        * 12.1. Registration (2)
5983
5984      #credential-key-pairReferenced in:
5985        * 4. Terminology (2) (3)
5986
5987      #credential-private-keyReferenced in:
5988        * 4. Terminology (2) (3) (4) (5) (6)
5989        * 5.1. PublicKeyCredential Interface
5990        * 5.2.2. Web Authentication Assertion (interface
5991          AuthenticatorAssertionResponse)
5992        * 6. WebAuthn Authenticator model
5993        * 6.2.2. The authenticatorGetAssertion operation
5994        * 6.3. Attestation (2)
5995        * 7.2. Verifying an authentication assertion
5996
5997      #public-key-credential-sourceReferenced in:
5998        * 4. Terminology (2) (3) (4) (5) (6) (7) (8)
5999        * 5.1.3. Create a new credential - PublicKeyCredential's
6000          [[Create]](origin, options, sameOriginWithAncestors) method
6001
6002      #public-key-credential-source-managing-authenticatorReferenced in:
6003        * 4. Terminology
6004
6005      #public-key-credentialReferenced in:
6006        * 1. Introduction (2) (3) (4) (5)
6007        * 4. Terminology (2) (3) (4) (5) (6) (7) (8)
6008        * 5. Web Authentication API (2) (3) (4)
6009        * 5.1. PublicKeyCredential Interface
6010        * 5.1.3. Create a new credential - PublicKeyCredential's
6011          [[Create]](origin, options, sameOriginWithAncestors) method
6012        * 5.1.4. Use an existing credential to make an assertion -
6013          PublicKeyCredential's [[Get]](options) method
6014        * 5.1.4.1. PublicKeyCredential's
6015          [[DiscoverFromExternalSource]](origin, options,
6016          sameOriginWithAncestors) method (2)
6017        * 5.2.1. Information about Public Key Credential (interface
6018          AuthenticatorAttestationResponse)
6019        * 5.4.1. Public Key Entity Description (dictionary
6020          PublicKeyCredentialEntity)
6021        * 5.4.4. Authenticator Selection Criteria (dictionary
6022          AuthenticatorSelectionCriteria)
```

#registrationReferenced in:
* 1. Introduction (2)
* 3. Terminology (2) (3) (4) (5) (6) (7) (8) (9)
* 6.1. Registering a new credential

#relying-partyReferenced in:
* 1. Introduction (2) (3) (4) (5) (6) (7)
* 1.1.3. Other use cases and configurations
* 3. Terminology (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13) (14) (15) (16) (17) (18) (19) (20) (21) (22)
* 4. Web Authentication API (2) (3) (4) (5) (6) (7)
* 4.1.4. Use an existing credential to make an assertion - PublicKeyCredential's [[DiscoverFromExternalSource]](options) method (2)
* 4.1.5. Platform Authenticator Availability - PublicKeyCredential's isPlatformAuthenticatorAvailable() method (2) (3)
* 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)
* 4.4. Options for Credential Creation (dictionary MakePublicKeyCredentialOptions) (2) (3) (4) (5) (6) (7) (8)
* 4.4.1. Public Key Entity Description (dictionary PublicKeyCredentialEntity) (2) (3) (4) (5)
* 4.4.3. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria) (2) (3)
* 4.4.4. Authenticator Attachment enumeration (enum AuthenticatorAttachment) (2) (3) (4)
* 4.7.1. Client data used in WebAuthn signatures (dictionary CollectedClientData) (2) (3) (4)
* 4.7.4. Authenticator Transport enumeration (enum AuthenticatorTransport) (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. Attestation (2) (3) (4) (5) (6)
* 5.3.5.1. Privacy
* 5.3.5.2. Attestation Certificate and Attestation Certificate CA

#registrationReferenced in:
* 1. Introduction (2)
* 4. Terminology (2) (3) (4) (5) (6) (7) (8) (9)
* 7.1. Registering a new credential

#relying-partyReferenced in:
* 1. Introduction (2) (3) (4) (5) (6) (7)
* 1.1.3. Other use cases and configurations
* 2.3. Relying Parties
* 4. Terminology (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13) (14) (15) (16) (17) (18) (19) (20) (21) (22) (23) (24) (25) (26) (27) (28) (29) (30)
* 5. Web Authentication API (2) (3) (4) (5) (6) (7)
* 5.1. PublicKeyCredential Interface (2)
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
* 5.1.4. Use an existing credential to make an assertion - PublicKeyCredential's [[Get]](options) method (2)
* 5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method (2) (3) (4)
* 5.1.6. Availability of User-Verifying Platform Authenticator - PublicKeyCredential's isUserVerifyingPlatformAuthenticatorAvailable() method (2) (3)
* 5.2. Authenticator Responses (interface AuthenticatorResponse)
* 5.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse) (2)
* 5.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
* 5.4. Options for Credential Creation (dictionary MakePublicKeyCredentialOptions) (2) (3) (4) (5) (6) (7)
* 5.4.1. Public Key Entity Description (dictionary PublicKeyCredentialEntity) (2) (3)
* 5.4.2. RP Parameters for Credential Generation (dictionary PublicKeyCredentialRpEntity) (2)
* 5.4.4. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria) (2) (3)
* 5.4.5. Authenticator Attachment enumeration (enum AuthenticatorAttachment) (2) (3) (4)
* 5.4.6. Attestation Conveyance Preference enumeration (enum AttestationConveyancePreference) (2) (3) (4) (5) (6) (7)
* 5.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)
* 5.8.1. Client data used in WebAuthn signatures (dictionary CollectedClientData) (2) (3) (4)
* 5.8.4. Authenticator Transport enumeration (enum AuthenticatorTransport) (2)
* 5.8.6. User Verification Requirement enumeration (enum UserVerificationRequirement) (2) (3) (4)
* 6. WebAuthn Authenticator model (2)
* 6.1. Authenticator data (2)
* 6.1.1. Signature Counter Considerations (2) (3) (4) (5) (6)
* 6.2.1. The authenticatorMakeCredential operation (2) (3) (4) (5) (6)
* 6.2.2. The authenticatorGetAssertion operation (2) (3)
* 6.3. Attestation (2) (3) (4) (5) (6)
* 5.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)
* 5.8. Supporting Data Structures
* 6. WebAuthn Authenticator model (2) (3) (4) (5)
* 6.2.2. The authenticatorGetAssertion operation (2) (3)
* 6.3. Attestation (2)
* 6.3.2. Attestation Statement Formats
* 6.3.3. Attestation Types
* 6.3.5.2. Attestation Certificate and Attestation Certificate CA Compromise (2)
* 7.1. Registering a new credential
* 9. WebAuthn Extensions (2)
* 12. Sample scenarios

#registrationReferenced in:
* 1. Introduction (2)
* 4. Terminology (2) (3) (4) (5) (6) (7) (8) (9)
* 7.1. Registering a new credential

#relying-partyReferenced in:
* 1. Introduction (2) (3) (4) (5) (6) (7)
* 1.1.3. Other use cases and configurations
* 2.3. Relying Parties
* 4. Terminology (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13) (14) (15) (16) (17) (18) (19) (20) (21) (22) (23) (24) (25) (26) (27) (28) (29) (30)
* 5. Web Authentication API (2) (3) (4) (5) (6) (7)
* 5.1. PublicKeyCredential Interface (2)
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
* 5.1.4. Use an existing credential to make an assertion - PublicKeyCredential's [[Get]](options) method (2)
* 5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method (2) (3) (4)
* 5.1.6. Availability of User-Verifying Platform Authenticator - PublicKeyCredential's isUserVerifyingPlatformAuthenticatorAvailable() method (2) (3)
* 5.2. Authenticator Responses (interface AuthenticatorResponse)
* 5.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse) (2)
* 5.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
* 5.4. Options for Credential Creation (dictionary MakePublicKeyCredentialOptions) (2) (3) (4) (5) (6) (7)
* 5.4.1. Public Key Entity Description (dictionary PublicKeyCredentialEntity) (2) (3)
* 5.4.2. RP Parameters for Credential Generation (dictionary PublicKeyCredentialRpEntity) (2)
* 5.4.4. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria) (2) (3)
* 5.4.5. Authenticator Attachment enumeration (enum AuthenticatorAttachment) (2) (3) (4)
* 5.4.6. Attestation Conveyance Preference enumeration (enum AttestationConveyancePreference) (2) (3) (4) (5) (6) (7)
* 5.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)
* 5.8.1. Client data used in WebAuthn signatures (dictionary CollectedClientData) (2) (3) (4)
* 5.8.4. Authenticator Transport enumeration (enum AuthenticatorTransport) (2)
* 5.8.6. User Verification Requirement enumeration (enum UserVerificationRequirement) (2) (3) (4)
* 6. WebAuthn Authenticator model (2)
* 6.1. Authenticator data (2)
* 6.1.1. Signature Counter Considerations (2) (3) (4) (5) (6)
* 6.2.1. The authenticatorMakeCredential operation (2) (3) (4) (5) (6)
* 6.2.2. The authenticatorGetAssertion operation (2) (3)
* 6.3. Attestation (2) (3) (4) (5) (6)

Compromise (2) (3) (4) (5) (6)
* 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) (5)
* 7.4. Android Key Attestation Statement Format
* 8. WebAuthn Extensions (2) (3) (4)
* 8.2. Defining extensions (2)
* 8.3. Extending request parameters (2) (3) (4)
* 8.6. Example Extension (2) (3)
* 9.1. FIDO AppId Extension (appid) (2)
* 9.2. Simple Transaction Authorization Extension (txAuthSimple)
* 9.4. Authenticator Selection Extension (authnSel) (2) (3)
* 9.5. Supported Extensions Extension (exts) (2)
* 9.6. User Verification Index Extension (uvi)
* 9.7. Location Extension (loc) (2)
* 10.2. WebAuthn Extension Identifier Registrations (2)
* 11.1. Registration (2) (3) (4) (5)
* 11.2. Registration Specifically with Platform Authenticator (2) (3)
* 11.3. Authentication (2) (3) (4) (5)
* 11.4. Decommissioning (2)

#relying-party-identifierReferenced in:
* 4. Web Authentication API
* 4.4. Options for Credential Creation (dictionary MakePublicKeyCredentialOptions)
* 4.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)
* 5. WebAuthn Authenticator model

#rp-idReferenced in:
* 3. Terminology (2) (3) (4) (5) (6)
* 4. Web Authentication API (2) (3) (4) (5)
* 4.1.3. Create a new credential - PublicKeyCredential's [[Create]](options) method (2)
* 4.1.4. Use an existing credential to make an assertion - PublicKeyCredential's [[DiscoverFromExternalSource]](options) method (2)
* 4.4.1. Public Key Entity Description (dictionary PublicKeyCredentialEntity)
* 5. WebAuthn Authenticator model
* 5.1. Authenticator data (2) (3) (4) (5) (6)
* 5.2.1. The authenticatorMakeCredential operation (2) (3)
* 5.2.2. The authenticatorGetAssertion operation (2) (3)
* 6.1. Registering a new credential (2)
* 6.2. Verifying an authentication assertion (2)
* 7.4. Android Key Attestation Statement Format
* 7.6. FIDO U2F Attestation Statement Format (2) (3)

#public-key-credentialReferenced in:
* 1. Introduction (2) (3) (4) (5)
* 3. Terminology (2) (3) (4) (5) (6) (7) (8)
* 4. Web Authentication API (2) (3) (4)
* 4.1. PublicKeyCredential Interface
* 4.1.3. Create a new credential - PublicKeyCredential's [[Create]](options) method
* 4.1.4. Use an existing credential to make an assertion - PublicKeyCredential's [[DiscoverFromExternalSource]](options) method (2)
* 4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)
* 4.4.1. Public Key Entity Description (dictionary PublicKeyCredentialEntity)
* 4.4.3. Authenticator Selection Criteria (dictionary

* 6.3.5.1. Privacy
* 6.3.5.2. Attestation Certificate and Attestation Certificate CA Compromise (2) (3) (4) (5) (6)
* 7. Relying Party Operations (2) (3) (4)
* 7.1. Registering a new credential (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12)
* 7.2. Verifying an authentication assertion (2) (3) (4) (5) (6) (7) (8)
* 8.4. Android Key Attestation Statement Format
* 9. WebAuthn Extensions (2) (3) (4)
* 9.2. Defining extensions (2)
* 9.3. Extending request parameters (2) (3) (4)
* 9.6. Example Extension (2) (3)
* 10.1. FIDO AppId Extension (appid) (2)
* 10.2. Simple Transaction Authorization Extension (txAuthSimple)
* 10.4. Authenticator Selection Extension (authnSel) (2) (3)
* 10.5. Supported Extensions Extension (exts) (2)
* 10.6. User Verification Index Extension (uvi)
* 10.7. Location Extension (loc) (2)
* 11.2. WebAuthn Extension Identifier Registrations (2)
* 12.1. Registration (2) (3) (4) (5)
* 12.2. Registration Specifically with User Verifying Platform Authenticator (2) (3)
* 12.3. Authentication (2) (3) (4) (5)
* 12.5. Decommissioning (2)
* 13.1. Cryptographic Challenges

#relying-party-identifierReferenced in:
* 4. Terminology
* 5. Web Authentication API
* 5.4. Options for Credential Creation (dictionary MakePublicKeyCredentialOptions)
* 5.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)
* 6. WebAuthn Authenticator model

#rp-idReferenced in:
* 4. Terminology (2) (3) (4) (5)
* 5. Web Authentication API (2) (3) (4) (5)
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2)
* 5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method (2)
* 5.4.2. RP Parameters for Credential Generation (dictionary PublicKeyCredentialRpEntity)
* 6. WebAuthn Authenticator model
* 6.1. Authenticator data (2) (3) (4) (5) (6)
* 6.1.1. Signature Counter Considerations
* 6.2.1. The authenticatorMakeCredential operation (2)
* 6.2.2. The authenticatorGetAssertion operation (2)
* 7.1. Registering a new credential (2)
* 7.2. Verifying an authentication assertion
* 8.4. Android Key Attestation Statement Format
* 8.6. FIDO U2F Attestation Statement Format

```
5055      AuthenticatorSelectionCriteria)
5056    * 4.5. Options for Assertion Generation (dictionary
5057      PublicKeyCredentialRequestOptions)
5058    * 4.7. Supporting Data Structures
5059    * 5. WebAuthn Authenticator model (2) (3) (4) (5)
5060    * 5.2.2. The authenticatorGetAssertion operation (2) (3)
5061    * 5.3. Attestation (2)
5062    * 5.3.2. Attestation Statement Formats
5063    * 5.3.3. Attestation Types
5064    * 5.3.4. Generating an Attestation Object
5065    * 5.3.5.2. Attestation Certificate and Attestation Certificate CA
5066      Compromise (2)
5067    * 6.1. Registering a new credential
5068    * 8. WebAuthn Extensions (2)
5069    * 11. Sample scenarios
5070
5071  #test-of-user-presenceReferenced in:
5072    * 3. Terminology (2) (3) (4) (5) (6)
5073    * 9.2. Simple Transaction Authorization Extension (txAuthSimple)
5074    * 9.3. Generic Transaction Authorization Extension (txAuthGeneric)
5075
5076  #user-consentReferenced in:
5077    * 1. Introduction
5078    * 3. Terminology (2)
5079    * 4.1.3. Create a new credential - PublicKeyCredential's
5080      [[Create]](options) method
5081    * 4.2.2. Web Authentication Assertion (interface
5082      AuthenticatorAssertionResponse)
5083    * 5. WebAuthn Authenticator model (2) (3)
5084    * 5.2.2. The authenticatorGetAssertion operation (2)
5085
5086  #user-verificationReferenced in:
5087    * 1. Introduction
5088    * 3. Terminology (2) (3) (4) (5) (6) (7) (8)
5089    * 4.1.3. Create a new credential - PublicKeyCredential's
5090      [[Create]](options) method
5091    * 9.2. Simple Transaction Authorization Extension (txAuthSimple)
5092    * 9.3. Generic Transaction Authorization Extension (txAuthGeneric)
```

```
6148
6149  #test-of-user-presenceReferenced in:
6150    * 4. Terminology (2) (3) (4) (5) (6)
6151    * 6.2.1. The authenticatorMakeCredential operation
6152    * 6.2.2. The authenticatorGetAssertion operation
6153    * 10.2. Simple Transaction Authorization Extension (txAuthSimple)
6154    * 10.3. Generic Transaction Authorization Extension (txAuthGeneric)
6155
6156  #user-consentReferenced in:
6157    * 1. Introduction (2)
6158    * 4. Terminology (2)
6159    * 5. Web Authentication API
6160    * 5.1.4. Use an existing credential to make an assertion -
6161      PublicKeyCredential's [[Get]](options) method
6162    * 5.2.2. Web Authentication Assertion (interface
6163      AuthenticatorAssertionResponse)
6164    * 5.4.6. Attestation Conveyance Preference enumeration (enum
6165      AttestationConveyancePreference)
6166    * 6. WebAuthn Authenticator model (2) (3)
6167    * 6.2.1. The authenticatorMakeCredential operation (2) (3) (4) (5)
6168      (6)
6169    * 6.2.2. The authenticatorGetAssertion operation (2) (3) (4) (5)
6170    * 11.2. WebAuthn Extension Identifier Registrations
6171
6172  #user-handleReferenced in:
6173    * 4. Terminology
6174    * 5.1.4.1. PublicKeyCredential's
6175      [[DiscoverFromExternalSource]](origin, options,
6176      sameOriginWithAncestors) method
6177    * 5.2.2. Web Authentication Assertion (interface
6178      AuthenticatorAssertionResponse)
6179    * 5.4. Options for Credential Creation (dictionary
6180      MakePublicKeyCredentialOptions)
6181    * 5.4.3. User Account Parameters for Credential Generation
6182      (dictionary PublicKeyCredentialUserEntity)
6183    * 6.2.1. The authenticatorMakeCredential operation
6184    * 6.2.2. The authenticatorGetAssertion operation
6185
6186  #user-verificationReferenced in:
6187    * 1. Introduction
6188    * 4. Terminology (2) (3) (4) (5) (6) (7) (8) (9)
6189    * 5.1.3. Create a new credential - PublicKeyCredential's
6190      [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
6191    * 5.1.4.1. PublicKeyCredential's
6192      [[DiscoverFromExternalSource]](origin, options,
6193      sameOriginWithAncestors) method (2) (3)
6194    * 5.1.6. Availability of User-Verifying Platform Authenticator -
6195      PublicKeyCredential's
6196      isUserVerifyingPlatformAuthenticatorAvailable() method (2) (3) (4)
6197      (5)
6198    * 5.4.4. Authenticator Selection Criteria (dictionary
6199      AuthenticatorSelectionCriteria)
6200    * 5.5. Options for Assertion Generation (dictionary
6201      PublicKeyCredentialRequestOptions)
6202    * 5.8.6. User Verification Requirement enumeration (enum
```

```
5093
5094    #concept-user-presentReferenced in:
5095      * 3. Terminology
5096      * 5.1. Authenticator data (2) (3)
5097
5098    #upReferenced in:
5099      * 5.1. Authenticator data
5100
5101    #concept-user-verifiedReferenced in:
5102      * 3. Terminology
5103      * 5.1. Authenticator data (2) (3)
5104
5105    #uvReferenced in:
5106      * 5.1. Authenticator data
5107
5108    #webauthn-clientReferenced in:
5109      * 3. Terminology (2)
5110
5111    #web-authentication-apiReferenced in:
5112      * 1. Introduction (2) (3)
5113      * 3. Terminology (2)
5114
5115    #publickeycredentialReferenced in:
5116      * 1. Introduction
5117      * 4.1. PublicKeyCredential Interface (2) (3) (4) (5) (6) (7) (8)
5118      * 4.1.3. Create a new credential - PublicKeyCredential's
5119        [[Create]](options) method (2) (3) (4) (5) (6)
5120      * 4.1.4. Use an existing credential to make an assertion -
5121        PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5122        method (2) (3)
5123      * 4.1.5. Platform Authenticator Availability - PublicKeyCredential's
5124        isPlatformAuthenticatorAvailable() method
5125      * 4.7.3. Credential Descriptor (dictionary
5126        PublicKeyCredentialDescriptor)
5127      * 5.2.1. The authenticatorMakeCredential operation
5128      * 6. Relying Party Operations
5129      * 6.2. Verifying an authentication assertion
5130
5131    #dom-publickeycredential-rawidReferenced in:
5132      * 4.1. PublicKeyCredential Interface
5133      * 6.2. Verifying an authentication assertion
5134
5135    #dom-publickeycredential-responseReferenced in:
5136      * 4.1. PublicKeyCredential Interface
5137      * 4.1.3. Create a new credential - PublicKeyCredential's
5138        [[Create]](options) method
5139      * 4.1.4. Use an existing credential to make an assertion -
5140        PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5141        method
5142      * 6.2. Verifying an authentication assertion
5143
5144    #dom-publickeycredential-clientextensionresultsReferenced in:
5145      * 4.1. PublicKeyCredential Interface
5146      * 4.1.3. Create a new credential - PublicKeyCredential's
5147        [[Create]](options) method
5148      * 4.1.4. Use an existing credential to make an assertion -
```

```
6203      UserVerificationRequirement) (2) (3) (4)
6204      * 6.2.1. The authenticatorMakeCredential operation (2) (3)
6205      * 6.2.2. The authenticatorGetAssertion operation (2) (3)
6206      * 10.2. Simple Transaction Authorization Extension (txAuthSimple)
6207      * 10.3. Generic Transaction Authorization Extension (txAuthGeneric)
6208      * 12.2. Registration Specifically with User Verifying Platform
6209        Authenticator
6210
6211    #concept-user-presentReferenced in:
6212      * 4. Terminology
6213      * 6.1. Authenticator data (2) (3)
6214
6215    #upReferenced in:
6216      * 6.1. Authenticator data
6217
6218    #concept-user-verifiedReferenced in:
6219      * 4. Terminology
6220      * 6.1. Authenticator data (2) (3)
6221
6222    #uvReferenced in:
6223      * 5.8.6. User Verification Requirement enumeration (enum
6224        UserVerificationRequirement) (2)
6225      * 6.1. Authenticator data
6226
6227    #webauthn-clientReferenced in:
6228      * 4. Terminology (2)
6229      * 6.2.1. The authenticatorMakeCredential operation
6230      * 6.2.2. The authenticatorGetAssertion operation
6231
6232    #web-authentication-apiReferenced in:
6233      * 1. Introduction (2) (3)
6234      * 4. Terminology (2)
6235
6236    #publickeycredentialReferenced in:
6237      * 1. Introduction
6238      * 5.1. PublicKeyCredential Interface (2) (3) (4) (5) (6) (7) (8)
6239      * 5.1.3. Create a new credential - PublicKeyCredential's
6240        [[Create]](origin, options, sameOriginWithAncestors) method (2)
6241      * 5.1.4.1. PublicKeyCredential's
6242        [[DiscoverFromExternalSource]](origin, options,
6243        sameOriginWithAncestors) method (2)
6244      * 5.1.5. Store an existing credential - PublicKeyCredential's
6245        [[Store]](credential, sameOriginWithAncestors) method (2)
6246      * 5.1.6. Availability of User-Verifying Platform Authenticator -
6247        PublicKeyCredential's
6248        isUserVerifyingPlatformAuthenticatorAvailable() method
6249      * 5.8.3. Credential Descriptor (dictionary
6250        PublicKeyCredentialDescriptor)
6251      * 7. Relying Party Operations
6252      * 7.2. Verifying an authentication assertion
6253
6254    #dom-publickeycredential-rawidReferenced in:
6255      * 5.1. PublicKeyCredential Interface
6256      * 7.2. Verifying an authentication assertion
6257
6258    #dom-publickeycredential-getclientextensionresultsReferenced in:
6259      * 5.1. PublicKeyCredential Interface
6260      * 9.4. Client extension processing
6261
6262    #dom-publickeycredential-responseReferenced in:
6263      * 5.1. PublicKeyCredential Interface
6264      * 5.1.3. Create a new credential - PublicKeyCredential's
6265        [[Create]](origin, options, sameOriginWithAncestors) method
6266      * 5.1.4.1. PublicKeyCredential's
```

PublicKeyCredential's [[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 to make an assertion - PublicKeyCredential's [[DiscoverFromExternalSource]](options) method

#dom-credentialcreationoptions-publickeyReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's [[Create]](options) method (2) (3)

#dom-credentialrequestoptions-publickeyReferenced in:
* 4.1.4. Use an existing credential to make an assertion - PublicKeyCredential's [[DiscoverFromExternalSource]](options) method (2) (3)

#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

[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method
* 7.2. Verifying an authentication assertion

#dom-publickeycredential-identifier-slotReferenced in:
* 5.1. PublicKeyCredential Interface (2)
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
* 5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method

#dom-publickeycredential-clientextensionsresults-slotReferenced in:
* 5.1. PublicKeyCredential Interface
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
* 5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method

#dom-credentialcreationoptions-publickeyReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)

#dom-credentialrequestoptions-publickeyReferenced in:
* 5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method (2) (3)

#dom-publickeycredential-create-slotReferenced in:
* 5.1. PublicKeyCredential Interface
* 5.6. Abort operations with AbortSignal (2) (3) (4) (5)
* 6.2.1. The authenticatorMakeCredential operation

#dom-publickeycredential-create-origin-options-sameoriginwithancestors-originReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method

#dom-publickeycredential-create-origin-options-sameoriginwithancestors-optionsReferenced in:
* 7.1. Registering a new credential

#effective-user-verification-requirement-for-credential-creationReferenced in:
* 6.2.1. The authenticatorMakeCredential operation

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

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

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

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

#dom-publickeycredential-collectfromcredentialstore-slotReferenced in:
* 5.1.4. Use an existing credential to make an assertion - PublicKeyCredential's [[Get]](options) method

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

```
#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 to make an assertion -
    PublicKeyCredential's [[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)
  * 4.1.4. Use an existing credential to make an assertion -
    PublicKeyCredential's [[Get]](options) method
  * 5.6. Abort operations with AbortSignal (2) (3) (4) (5)
  * 6.2.2. The authenticatorGetAssertion operation

#dom-publickeycredential-discoverfromexternalsource-origin-options-same
originwithancestors-originReferenced in:
  * 5.1.4.1. PublicKeyCredential's
    [[DiscoverFromExternalSource]](origin, options,
    sameOriginWithAncestors) method

#effective-user-verification-requirement-for-assertionReferenced in:
  * 6.2.2. The authenticatorGetAssertion operation

#assertioncreationdata-credentialidresultReferenced in:
  * 5.1.4.1. PublicKeyCredential's
    [[DiscoverFromExternalSource]](origin, options,
    sameOriginWithAncestors) method (2) (3)

#assertioncreationdata-clientdatajsonresultReferenced in:
  * 5.1.4.1. PublicKeyCredential's
    [[DiscoverFromExternalSource]](origin, options,
    sameOriginWithAncestors) method

#assertioncreationdata-authenticatordataresultReferenced in:
  * 5.1.4.1. PublicKeyCredential's
    [[DiscoverFromExternalSource]](origin, options,
    sameOriginWithAncestors) method

#assertioncreationdata-signatureresultReferenced in:
  * 5.1.4.1. PublicKeyCredential's
    [[DiscoverFromExternalSource]](origin, options,
    sameOriginWithAncestors) method

#assertioncreationdata-userhandleresultReferenced in:
  * 5.1.4.1. PublicKeyCredential's
    [[DiscoverFromExternalSource]](origin, options,
    sameOriginWithAncestors) method

#assertioncreationdata-clientextensionresultsReferenced in:
  * 5.1.4.1. PublicKeyCredential's
    [[DiscoverFromExternalSource]](origin, options,
    sameOriginWithAncestors) method

#authenticatorresponseReferenced in:
  * 5.1. PublicKeyCredential Interface (2)
  * 5.2. Authenticator Responses (interface AuthenticatorResponse) (2)
  * 5.2.1. Information about Public Key Credential (interface
    AuthenticatorAttestationResponse) (2)
  * 5.2.2. Web Authentication Assertion (interface
    AuthenticatorAssertionResponse) (2)

#dom-authenticatorresponse-clientdatajsonReferenced in:
  * 5.1.3. Create a new credential - PublicKeyCredential's
    [[Create]](origin, options, sameOriginWithAncestors) method (2)
  * 5.1.4.1. PublicKeyCredential's
    [[DiscoverFromExternalSource]](origin, options,
    sameOriginWithAncestors) method (2)
  * 5.2. Authenticator Responses (interface AuthenticatorResponse)
  * 5.2.1. Information about Public Key Credential (interface
    AuthenticatorAttestationResponse)
  * 5.2.2. Web Authentication Assertion (interface
    AuthenticatorAssertionResponse)
  * 7.1. Registering a new credential (2)
  * 7.2. Verifying an authentication assertion

#authenticatorattestationresponseReferenced in:
  * 5.1. PublicKeyCredential Interface
  * 5.1.3. Create a new credential - PublicKeyCredential's
    [[Create]](origin, options, sameOriginWithAncestors) method
```

```
5205      * 4.2.1. Information about Public Key Credential (interface
5206        AuthenticatorAttestationResponse) (2)
5207      * 6. Relying Party Operations
5208      * 6.1. Registering a new credential (2) (3)
5209
5210    #dom-authenticatorattestationresponse-attestationobjectReferenced in:
5211      * 4.1.3. Create a new credential - PublicKeyCredential's
5212        [[Create]](options) method
5213      * 4.2.1. Information about Public Key Credential (interface
5214        AuthenticatorAttestationResponse)
5215      * 6.1. Registering a new credential
5216
5217    #authenticatorassertionresponseReferenced in:
5218      * 3. Terminology
5219      * 4.1. PublicKeyCredential Interface
5220      * 4.1.4. Use an existing credential to make an assertion -
5221        PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5222        method
5223      * 4.2.2. Web Authentication Assertion (interface
5224        AuthenticatorAssertionResponse) (2)
5225      * 6. Relying Party Operations
5226
5227    #dom-authenticatorassertionresponse-authenticatordataReferenced in:
5228      * 4.1.4. Use an existing credential to make an assertion -
5229        PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5230        method (2)
5231      * 4.2.2. Web Authentication Assertion (interface
5232        AuthenticatorAssertionResponse)
5233      * 6.2. Verifying an authentication assertion
5234
5235    #dom-authenticatorassertionresponse-signatureReferenced in:
5236      * 4.1.4. Use an existing credential to make an assertion -
5237        PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5238        method (2)
5239      * 4.2.2. Web Authentication Assertion (interface
                                                                                  AuthenticatorAssertionResponse)
5240      * 6.2. Verifying an authentication assertion
5241
5242
5243    #dictdef-publickeycredentialparametersReferenced in:
5244      * 4.3. Parameters for Credential Generation (dictionary
5245        PublicKeyCredentialParameters)
5246      * 4.4. Options for Credential Creation (dictionary
5247        MakePublicKeyCredentialOptions) (2)
5248
5249    #dom-publickeycredentialparameters-typeReferenced in:
5250      * 4.1.3. Create a new credential - PublicKeyCredential's
5251        [[Create]](options) method (2)
5252      * 4.3. Parameters for Credential Generation (dictionary
5253        PublicKeyCredentialParameters)
5254
5255    #dom-publickeycredentialparameters-algReferenced in:
5256      * 4.1.3. Create a new credential - PublicKeyCredential's
5257        [[Create]](options) method
5258      * 4.3. Parameters for Credential Generation (dictionary
5259        PublicKeyCredentialParameters)
5260
5261    #dictdef-makepublickeycredentialoptionsReferenced in:
5262      * 4.1.1. CredentialCreationOptions Extension
5263      * 4.1.3. Create a new credential - PublicKeyCredential's
5264        [[Create]](options) method
5265      * 4.4. Options for Credential Creation (dictionary
5266        MakePublickeyCredentialOptions)
```

```
6407      * 5.2.1. Information about Public Key Credential (interface
6408        AuthenticatorAttestationResponse) (2)
6409      * 7. Relying Party Operations
6410      * 7.1. Registering a new credential (2) (3)
6411
6412    #dom-authenticatorattestationresponse-attestationobjectReferenced in:
6413      * 5.1.3. Create a new credential - PublicKeyCredential's
6414        [[Create]](origin, options, sameOriginWithAncestors) method
6415      * 5.2.1. Information about Public Key Credential (interface
6416        AuthenticatorAttestationResponse)
6417      * 7.1. Registering a new credential
6418
6419    #authenticatorassertionresponseReferenced in:
6420      * 4. Terminology
6421      * 5.1. PublicKeyCredential Interface
6422      * 5.1.4.1. PublicKeyCredential's
6423        [[DiscoverFromExternalSource]](origin, options,
6424        sameOriginWithAncestors) method
6425      * 5.2.2. Web Authentication Assertion (interface
6426        AuthenticatorAssertionResponse) (2)
6427      * 7. Relying Party Operations
6428
6429    #dom-authenticatorassertionresponse-authenticatordataReferenced in:
6430      * 5.1.4.1. PublicKeyCredential's
6431        [[DiscoverFromExternalSource]](origin, options,
6432        sameOriginWithAncestors) method
6433      * 5.2.2. Web Authentication Assertion (interface
6434        AuthenticatorAssertionResponse)
6435      * 7.2. Verifying an authentication assertion
6436
6437    #dom-authenticatorassertionresponse-signatureReferenced in:
6438      * 5.1.4.1. PublicKeyCredential's
6439        [[DiscoverFromExternalSource]](origin, options,
6440        sameOriginWithAncestors) method
6441      * 5.2.2. Web Authentication Assertion (interface
6442        AuthenticatorAssertionResponse)
6443      * 7.2. Verifying an authentication assertion
6444
6445    #dom-authenticatorassertionresponse-userhandleReferenced in:
6446      * 5.1.4.1. PublicKeyCredential's
6447        [[DiscoverFromExternalSource]](origin, options,
6448        sameOriginWithAncestors) method
6449      * 5.2.2. Web Authentication Assertion (interface
6450        AuthenticatorAssertionResponse)
6451
6452    #dictdef-publickeycredentialparametersReferenced in:
6453      * 5.3. Parameters for Credential Generation (dictionary
6454        PublicKeyCredentialParameters)
6455      * 5.4. Options for Credential Creation (dictionary
6456        MakePublicKeyCredentialOptions) (2)
6457
6458    #dom-publickeycredentialparameters-typeReferenced in:
6459      * 5.1.3. Create a new credential - PublicKeyCredential's
6460        [[Create]](origin, options, sameOriginWithAncestors) method (2)
6461      * 5.3. Parameters for Credential Generation (dictionary
6462        PublicKeyCredentialParameters)
6463
6464    #dom-publickeycredentialparameters-algReferenced in:
6465      * 5.1.3. Create a new credential - PublicKeyCredential's
6466        [[Create]](origin, options, sameOriginWithAncestors) method
6467      * 5.3. Parameters for Credential Generation (dictionary
6468        PublicKeyCredentialParameters)
6469
6470    #dictdef-makepublickeycredentialoptionsReferenced in:
6471      * 5.1.1. CredentialCreationOptions Extension
6472      * 5.1.3. Create a new credential - PublicKeyCredential's
6473        [[Create]](origin, options, sameOriginWithAncestors) method
6474      * 5.4. Options for Credential Creation (dictionary
6475        MakePublicKeyCredentialOptions)
```

```
5267
5268      #dom-makepublickeycredentialoptions-rpReferenced in:
5269        * 4.1.3. Create a new credential - PublicKeyCredential's
5270          [[Create]](options) method (2) (3) (4) (5) (6)
5271        * 4.4. Options for Credential Creation (dictionary
5272          MakePublicKeyCredentialOptions)
5273
5274      #dom-makepublickeycredentialoptions-userReferenced in:
5275        * 4.1.3. Create a new credential - PublicKeyCredential's
5276          [[Create]](options) method (2) (3) (4)
5277        * 4.4. Options for Credential Creation (dictionary
5278          MakePublicKeyCredentialOptions)
5279        * 5.2.1. The authenticatorMakeCredential operation (2)
5280        * 6.1. Registering a new credential
5281
5282      #dom-makepublickeycredentialoptions-challengeReferenced in:
5283        * 4.1.3. Create a new credential - PublicKeyCredential's
5284          [[Create]](options) method
5285        * 4.4. Options for Credential Creation (dictionary
5286          MakePublicKeyCredentialOptions)
5287
5288      #dom-makepublickeycredentialoptions-pubkeycredparamsReferenced in:
5289        * 4.1.3. Create a new credential - PublicKeyCredential's
5290          [[Create]](options) method (2)
5291        * 4.4. Options for Credential Creation (dictionary
5292          MakePublicKeyCredentialOptions)
5293
5294      #dom-makepublickeycredentialoptions-timeoutReferenced in:
5295        * 4.1.3. Create a new credential - PublicKeyCredential's
5296          [[Create]](options) method (2)
5297        * 4.4. Options for Credential Creation (dictionary
5298          MakePublicKeyCredentialOptions)
5299
5300      #dom-makepublickeycredentialoptions-excludecredentialsReferenced in:
5301        * 4.1.3. Create a new credential - PublicKeyCredential's
5302          [[Create]](options) method
5303        * 4.4. Options for Credential Creation (dictionary
5304          MakePublicKeyCredentialOptions)
5305
5306      #dom-makepublickeycredentialoptions-authenticatorselectionReferenced
5307      in:
5308        * 4.1.3. Create a new credential - PublicKeyCredential's
5309          [[Create]](options) method (2)
5310        * 4.4. Options for Credential Creation (dictionary
5311          MakePublicKeyCredentialOptions)
5312        * 5.2.1. The authenticatorMakeCredential operation (2)
5313
5314      #dom-makepublickeycredentialoptions-extensionsReferenced in:
5315        * 4.1.3. Create a new credential - PublicKeyCredential's
5316          [[Create]](options) method (2)
5317        * 4.4. Options for Credential Creation (dictionary
5318          MakePublicKeyCredentialOptions)
5319        * 8.3. Extending request parameters
5320
5321      #dictdef-publickeycredentialentityReferenced in:
5322        * 4.4. Options for Credential Creation (dictionary
5323          MakePublicKeyCredentialOptions) (2)
5324        * 4.4.1. Public Key Entity Description (dictionary
5325          PublicKeyCredentialEntity) (2)
5326        * 4.4.2. User Account Parameters for Credential Generation
```

```
6476
6477      #dom-makepublickeycredentialoptions-rpReferenced in:
6478        * 5.1.3. Create a new credential - PublicKeyCredential's
6479          [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
6480          (4) (5) (6)
6481        * 5.4. Options for Credential Creation (dictionary
6482          MakePublicKeyCredentialOptions)
6483
6484      #dom-makepublickeycredentialoptions-userReferenced in:
6485        * 5.1.3. Create a new credential - PublicKeyCredential's
6486          [[Create]](origin, options, sameOriginWithAncestors) method
6487        * 5.4. Options for Credential Creation (dictionary
6488          MakePublicKeyCredentialOptions)
6489        * 7.1. Registering a new credential
6490
6491      #dom-makepublickeycredentialoptions-challengeReferenced in:
6492        * 5.1.3. Create a new credential - PublicKeyCredential's
6493          [[Create]](origin, options, sameOriginWithAncestors) method
6494        * 5.4. Options for Credential Creation (dictionary
6495          MakePublicKeyCredentialOptions)
6496
6497      #dom-makepublickeycredentialoptions-pubkeycredparamsReferenced in:
6498        * 5.1.3. Create a new credential - PublicKeyCredential's
6499          [[Create]](origin, options, sameOriginWithAncestors) method (2)
6500        * 5.4. Options for Credential Creation (dictionary
6501          MakePublicKeyCredentialOptions)
6502
6503      #dom-makepublickeycredentialoptions-timeoutReferenced in:
6504        * 5.1.3. Create a new credential - PublicKeyCredential's
6505          [[Create]](origin, options, sameOriginWithAncestors) method (2)
6506        * 5.4. Options for Credential Creation (dictionary
6507          MakePublicKeyCredentialOptions)
6508
6509      #dom-makepublickeycredentialoptions-excludecredentialsReferenced in:
6510        * 5.1.3. Create a new credential - PublicKeyCredential's
6511          [[Create]](origin, options, sameOriginWithAncestors) method
6512        * 5.4. Options for Credential Creation (dictionary
6513          MakePublicKeyCredentialOptions)
6514
6515      #dom-makepublickeycredentialoptions-authenticatorselectionReferenced
6516      in:
6517        * 5.1.3. Create a new credential - PublicKeyCredential's
6518          [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
6519          (4) (5) (6)
6520        * 5.4. Options for Credential Creation (dictionary
6521          MakePublicKeyCredentialOptions)
6522        * 6.2.1. The authenticatorMakeCredential operation
6523
6524      #dom-makepublickeycredentialoptions-attestationReferenced in:
6525        * 5.1.3. Create a new credential - PublicKeyCredential's
6526          [[Create]](origin, options, sameOriginWithAncestors) method
6527        * 5.4. Options for Credential Creation (dictionary
6528          MakePublicKeyCredentialOptions)
6529
6530      #dom-makepublickeycredentialoptions-extensionsReferenced in:
6531        * 5.1.3. Create a new credential - PublicKeyCredential's
6532          [[Create]](origin, options, sameOriginWithAncestors) method (2)
6533        * 5.4. Options for Credential Creation (dictionary
6534          MakePublicKeyCredentialOptions)
6535        * 9.3. Extending request parameters
6536
6537      #dictdef-publickeycredentialentityReferenced in:
6538        * 5.4.1. Public Key Entity Description (dictionary
6539          PublicKeyCredentialEntity) (2)
6540        * 5.4.2. RP Parameters for Credential Generation (dictionary
6541          PublicKeyCredentialRpEntity)
```

```
5327        (dictionary PublicKeyCredentialUserEntity)
5328      * 5.2.1. The authenticatorMakeCredential operation
5329
5330      #dom-publickeycredentialentity-idReferenced in:
5331        * 4.1.3. Create a new credential - PublicKeyCredential's
5332          [[Create]](options) method (2) (3) (4) (5)
5333        * 4.4. Options for Credential Creation (dictionary
5334          MakePublicKeyCredentialOptions) (2) (3)
5335        * 4.4.1. Public Key Entity Description (dictionary
5336          PublicKeyCredentialEntity)
5337        * 5.2.1. The authenticatorMakeCredential operation (2)
5338
5339      #dom-publickeycredentialentity-nameReferenced in:
5340        * 4.1.3. Create a new credential - PublicKeyCredential's
5341          [[Create]](options) method (2)
5342        * 4.4. Options for Credential Creation (dictionary
5343          MakePublicKeyCredentialOptions) (2)
5344        * 4.4.1. Public Key Entity Description (dictionary
5345          PublicKeyCredentialEntity)
5346
5347      #dom-publickeycredentialentity-iconReferenced in:
5348        * 4.4.1. Public Key Entity Description (dictionary
5349          PublicKeyCredentialEntity)
5350
5351      #dictdef-publickeycredentialuserentityReferenced in:
5352        * 4.4. Options for Credential Creation (dictionary
5353          MakePublicKeyCredentialOptions) (2)
5354        * 4.4.2. User Account Parameters for Credential Generation
5355          (dictionary PublicKeyCredentialUserEntity) (2)
5356        * 5.2.1. The authenticatorMakeCredential operation
5357
5358      #dom-publickeycredentialuserentity-displaynameReferenced in:
5359        * 4.1.3. Create a new credential - PublicKeyCredential's
5360          [[Create]](options) method
5361        * 4.4. Options for Credential Creation (dictionary
5362          MakePublicKeyCredentialOptions)
5363        * 4.4.2. User Account Parameters for Credential Generation
5364          (dictionary PublicKeyCredentialUserEntity)
5365
5366      #dictdef-authenticatorselectioncriteriaReferenced in:
5367        * 4.4. Options for Credential Creation (dictionary
5368          MakePublicKeyCredentialOptions) (2)
5369        * 4.4.3. Authenticator Selection Criteria (dictionary
5370          AuthenticatorSelectionCriteria) (2)
5371
5372      #dom-authenticatorselectioncriteria-aaReferenced in:
```

```
6542        * 5.4.3. User Account Parameters for Credential Generation
6543          (dictionary PublicKeyCredentialUserEntity)
6544
6545      #dom-publickeycredentialentity-nameReferenced in:
6546        * 5.4. Options for Credential Creation (dictionary
6547          MakePublicKeyCredentialOptions) (2)
6548        * 5.4.1. Public Key Entity Description (dictionary
6549          PublicKeyCredentialEntity)
6550        * 6.2.1. The authenticatorMakeCredential operation (2)
6551
6552      #dom-publickeycredentialentity-iconReferenced in:
6553        * 5.4.1. Public Key Entity Description (dictionary
6554          PublicKeyCredentialEntity)
6555
6556      #dictdef-publickeycredentialrpentityReferenced in:
6557        * 5.4. Options for Credential Creation (dictionary
6558          MakePublicKeyCredentialOptions) (2)
6559        * 5.4.2. RP Parameters for Credential Generation (dictionary
6560          PublicKeyCredentialRpEntity) (2)
6561        * 6.2.1. The authenticatorMakeCredential operation
6562
6563      #dom-publickeycredentialrpentity-idReferenced in:
6564        * 5.4. Options for Credential Creation (dictionary
6565          MakePublicKeyCredentialOptions)
6566        * 5.4.2. RP Parameters for Credential Generation (dictionary
6567          PublicKeyCredentialRpEntity)
6568        * 6.2.1. The authenticatorMakeCredential operation (2) (3) (4)
6569
6570      #dictdef-publickeycredentialuserentityReferenced in:
6571        * 5.4. Options for Credential Creation (dictionary
6572          MakePublicKeyCredentialOptions) (2)
6573        * 5.4.3. User Account Parameters for Credential Generation
6574          (dictionary PublicKeyCredentialUserEntity) (2)
6575        * 6.2.1. The authenticatorMakeCredential operation
6576
6577      #dom-publickeycredentialuserentity-idReferenced in:
6578        * 5.4. Options for Credential Creation (dictionary
6579          MakePublicKeyCredentialOptions)
6580        * 5.4.3. User Account Parameters for Credential Generation
6581          (dictionary PublicKeyCredentialUserEntity)
6582        * 6.2.1. The authenticatorMakeCredential operation
6583
6584      #dom-publickeycredentialuserentity-displaynameReferenced in:
6585        * 4. Terminology
6586        * 5.4. Options for Credential Creation (dictionary
6587          MakePublicKeyCredentialOptions)
6588        * 5.4.3. User Account Parameters for Credential Generation
6589          (dictionary PublicKeyCredentialUserEntity)
6590        * 6.2.1. The authenticatorMakeCredential operation
6591
6592      #dictdef-authenticatorselectioncriteriaReferenced in:
6593        * 5.4. Options for Credential Creation (dictionary
6594          MakePublicKeyCredentialOptions) (2)
6595        * 5.4.4. Authenticator Selection Criteria (dictionary
6596          AuthenticatorSelectionCriteria) (2)
6597
6598      #dom-authenticatorselectioncriteria-authenticatorattachmentReferenced
```

```
5373      * 4.1.3. Create a new credential - PublicKeyCredential's
5374        [[Create]](options) method
5375      * 4.4.3. Authenticator Selection Criteria (dictionary
            AuthenticatorSelectionCriteria)
5376
5377    #dom-authenticatorselectioncriteria-rkReferenced in:
5378      * 4.1.3. Create a new credential - PublicKeyCredential's
5379        [[Create]](options) method (2)
5380      * 4.4.3. Authenticator Selection Criteria (dictionary
5381        AuthenticatorSelectionCriteria)
5382
5383
5384    #dom-authenticatorselectioncriteria-uvReferenced in:
5385      * 4.1.3. Create a new credential - PublicKeyCredential's
5386        [[Create]](options) method
5387      * 4.4.3. Authenticator Selection Criteria (dictionary
5388        AuthenticatorSelectionCriteria)
5389
5390    #enumdef-authenticatorattachmentReferenced in:
5391      * 4.4.3. Authenticator Selection Criteria (dictionary
5392        AuthenticatorSelectionCriteria) (2)
5393      * 4.4.4. Authenticator Attachment enumeration (enum
5394        AuthenticatorAttachment) (2)
5395
5396    #platform-authenticatorsReferenced in:
5397      * 4.1.5. Platform Authenticator Availability - PublicKeyCredential's
5398        isPlatformAuthenticatorAvailable() method (2) (3) (4) (5)
5399      * 4.4.4. Authenticator Attachment enumeration (enum
5400        AuthenticatorAttachment) (2)
5401      * 11.1. Registration
5402      * 11.2. Registration Specifically with Platform Authenticator (2)
5403
5404    #roaming-authenticatorsReferenced in:
5405      * 1.1.3. Other use cases and configurations
5406      * 4.4.4. Authenticator Attachment enumeration (enum
5407        AuthenticatorAttachment) (2)
5408      * 11.1. Registration
5409
5410    #platform-attachmentReferenced in:
5411      * 4.4.4. Authenticator Attachment enumeration (enum
5412        AuthenticatorAttachment)
5413
5414    #cross-platform-attachedReferenced in:
5415      * 4.4.4. Authenticator Attachment enumeration (enum
5416        AuthenticatorAttachment) (2)
5417
```

```
6599    in:
6600      * 5.1.3. Create a new credential - PublicKeyCredential's
6601        [[Create]](origin, options, sameOriginWithAncestors) method
6602      * 5.4.4. Authenticator Selection Criteria (dictionary
6603        AuthenticatorSelectionCriteria)
6604
6605    #dom-authenticatorselectioncriteria-requireresidentkeyReferenced in:
6606      * 5.1.3. Create a new credential - PublicKeyCredential's
6607        [[Create]](origin, options, sameOriginWithAncestors) method (2)
6608      * 5.4.4. Authenticator Selection Criteria (dictionary
6609        AuthenticatorSelectionCriteria)
6610      * 6.2.1. The authenticatorMakeCredential operation
6611
6612    #dom-authenticatorselectioncriteria-userverificationReferenced in:
6613      * 5.1.3. Create a new credential - PublicKeyCredential's
6614        [[Create]](origin, options, sameOriginWithAncestors) method (2)
6615      * 5.4.4. Authenticator Selection Criteria (dictionary
6616        AuthenticatorSelectionCriteria)
6617
6618    #enumdef-authenticatorattachmentReferenced in:
6619      * 5.4.4. Authenticator Selection Criteria (dictionary
6620        AuthenticatorSelectionCriteria) (2)
6621      * 5.4.5. Authenticator Attachment enumeration (enum
6622        AuthenticatorAttachment) (2)
6623
6624    #platform-authenticatorsReferenced in:
6625      * 5.1.6. Availability of User-Verifying Platform Authenticator -
6626        PublicKeyCredential's
6627        isUserVerifyingPlatformAuthenticatorAvailable() method (2) (3) (4)
6628        (5)
6629      * 5.4.5. Authenticator Attachment enumeration (enum
6630        AuthenticatorAttachment) (2)
6631      * 12.1. Registration
6632      * 12.2. Registration Specifically with User Verifying Platform
6633        Authenticator (2)
6634
6635    #roaming-authenticatorsReferenced in:
6636      * 1.1.3. Other use cases and configurations
6637      * 5.4.5. Authenticator Attachment enumeration (enum
6638        AuthenticatorAttachment) (2)
6639      * 12.1. Registration
6640
6641    #platform-attachmentReferenced in:
6642      * 5.4.5. Authenticator Attachment enumeration (enum
6643        AuthenticatorAttachment)
6644
6645    #cross-platform-attachedReferenced in:
6646      * 5.4.5. Authenticator Attachment enumeration (enum
6647        AuthenticatorAttachment) (2)
6648
6649    #attestation-conveyanceReferenced in:
6650      * 4. Terminology
6651      * 5.4. Options for Credential Creation (dictionary
6652        MakePublicKeyCredentialOptions)
6653      * 5.4.6. Attestation Conveyance Preference enumeration (enum
6654        AttestationConveyancePreference)
6655
6656    #enumdef-attestationconveyancepreferenceReferenced in:
6657      * 5.4. Options for Credential Creation (dictionary
6658        MakePublicKeyCredentialOptions) (2)
6659      * 5.4.6. Attestation Conveyance Preference enumeration (enum
6660        AttestationConveyancePreference) (2)
6661
6662    #dom-attestationconveyancepreference-noneReferenced in:
6663      * 5.4. Options for Credential Creation (dictionary
6664        MakePublicKeyCredentialOptions)
6665      * 5.4.6. Attestation Conveyance Preference enumeration (enum
6666        AttestationConveyancePreference)
6667
6668    #dom-attestationconveyancepreference-indirectReferenced in:
```

```
5418    #dictdef-publickeycredentialrequestoptionsReferenced in:
5419     * 4.1.2. CredentialRequestOptions Extension
5420     * 4.5. Options for Assertion Generation (dictionary
5421       PublicKeyCredentialRequestOptions) (2)
5422     * 6.2. Verifying an authentication assertion
5423
5424    #dom-publickeycredentialrequestoptions-challengeReferenced in:
5425     * 4.1.4. Use an existing credential to make an assertion -
5426       PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5427       method
5428     * 4.5. Options for Assertion Generation (dictionary
5429       PublicKeyCredentialRequestOptions) (2)
5430
5431    #dom-publickeycredentialrequestoptions-timeoutReferenced in:
5432     * 4.1.4. Use an existing credential to make an assertion -
5433       PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5434       method (2)
5435     * 4.5. Options for Assertion Generation (dictionary
5436       PublicKeyCredentialRequestOptions)
5437
5438    #dom-publickeycredentialrequestoptions-rpidReferenced in:
5439     * 4.1.4. Use an existing credential to make an assertion -
5440       PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5441       method (2) (3) (4)
5442     * 4.5. Options for Assertion Generation (dictionary
5443       PublicKeyCredentialRequestOptions)
5444     * 9.1. FIDO AppId Extension (appid)
5445
5446    #dom-publickeycredentialrequestoptions-allowcredentialsReferenced in:
5447     * 4.1.4. Use an existing credential to make an assertion -
5448       PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5449       method (2) (3) (4)
5450     * 4.5. Options for Assertion Generation (dictionary
5451       PublicKeyCredentialRequestOptions)
5452
5453    #dom-publickeycredentialrequestoptions-extensionsReferenced in:
5454     * 4.1.4. Use an existing credential to make an assertion -
5455       PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5456       method (2)
5457     * 4.5. Options for Assertion Generation (dictionary
5458       PublicKeyCredentialRequestOptions)
5459
5460    #typedefdef-authenticationextensionsReferenced in:
5461     * 4.1. PublicKeyCredential Interface (2)
5462     * 4.1.3. Create a new credential - PublicKeyCredential's
5463       [[Create]](options) method
5464     * 4.1.4. Use an existing credential to make an assertion -
5465       PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5466       method
5467     * 4.4. Options for Credential Creation (dictionary
5468       MakePublicKeyCredentialOptions) (2)
5469     * 4.5. Options for Assertion Generation (dictionary
```

```
6669     * 5.4.6. Attestation Conveyance Preference enumeration (enum
6670       AttestationConveyancePreference)
6671
6672    #dom-attestationconveyancepreference-directReferenced in:
6673     * 5.4.6. Attestation Conveyance Preference enumeration (enum
6674       AttestationConveyancePreference)
6675
6676    #dictdef-publickeycredentialrequestoptionsReferenced in:
6677     * 5.1.2. CredentialRequestOptions Extension
6678     * 5.1.4.1. PublicKeyCredential's
6679       [[DiscoverFromExternalSource]](origin, options,
6680       sameOriginWithAncestors) method
6681     * 5.5. Options for Assertion Generation (dictionary
6682       PublicKeyCredentialRequestOptions) (2)
6683     * 7.2. Verifying an authentication assertion
6684
6685    #dom-publickeycredentialrequestoptions-challengeReferenced in:
6686     * 5.1.4.1. PublicKeyCredential's
6687       [[DiscoverFromExternalSource]](origin, options,
6688       sameOriginWithAncestors) method
6689     * 5.5. Options for Assertion Generation (dictionary
6690       PublicKeyCredentialRequestOptions) (2)
6691     * 13.1. Cryptographic Challenges
6692
6693    #dom-publickeycredentialrequestoptions-timeoutReferenced in:
6694     * 5.1.4.1. PublicKeyCredential's
6695       [[DiscoverFromExternalSource]](origin, options,
6696       sameOriginWithAncestors) method (2)
6697     * 5.5. Options for Assertion Generation (dictionary
6698       PublicKeyCredentialRequestOptions)
6699
6700    #dom-publickeycredentialrequestoptions-rpidReferenced in:
6701     * 5.1.4.1. PublicKeyCredential's
6702       [[DiscoverFromExternalSource]](origin, options,
6703       sameOriginWithAncestors) method (2) (3) (4)
6704     * 5.5. Options for Assertion Generation (dictionary
6705       PublicKeyCredentialRequestOptions)
6706     * 10.1. FIDO AppId Extension (appid)
6707
6708    #dom-publickeycredentialrequestoptions-allowcredentialsReferenced in:
6709     * 5.1.4.1. PublicKeyCredential's
6710       [[DiscoverFromExternalSource]](origin, options,
6711       sameOriginWithAncestors) method (2) (3) (4)
6712     * 5.5. Options for Assertion Generation (dictionary
6713       PublicKeyCredentialRequestOptions)
6714
6715    #dom-publickeycredentialrequestoptions-userverificationReferenced in:
6716     * 5.1.4.1. PublicKeyCredential's
6717       [[DiscoverFromExternalSource]](origin, options,
6718       sameOriginWithAncestors) method (2)
6719     * 5.5. Options for Assertion Generation (dictionary
6720       PublicKeyCredentialRequestOptions)
6721
6722    #dom-publickeycredentialrequestoptions-extensionsReferenced in:
6723     * 5.1.4.1. PublicKeyCredential's
6724       [[DiscoverFromExternalSource]](origin, options,
6725       sameOriginWithAncestors) method (2)
6726     * 5.5. Options for Assertion Generation (dictionary
6727       PublicKeyCredentialRequestOptions)
6728
6729    #typedefdef-authenticationextensionsReferenced in:
6730     * 5.1. PublicKeyCredential Interface
6731     * 5.1.3. Create a new credential - PublicKeyCredential's
6732       [[Create]](origin, options, sameOriginWithAncestors) method
6733     * 5.1.4.1. PublicKeyCredential's
6734       [[DiscoverFromExternalSource]](origin, options,
6735       sameOriginWithAncestors) method
6736     * 5.4. Options for Credential Creation (dictionary
6737       MakePublicKeyCredentialOptions) (2)
6738     * 5.5. Options for Assertion Generation (dictionary
```

```
5470      PublicKeyCredentialRequestOptions) (2)
5471    * 4.7.1. Client data used in WebAuthn signatures (dictionary
5472      CollectedClientData) (2)
5473
5474  #dictdef-collectedclientdataReferenced in:
5475    * 4.1.3. Create a new credential - PublicKeyCredential's
5476      [[Create]](options) method
5477    * 4.1.4. Use an existing credential to make an assertion -
5478      PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5479      method
5480    * 4.7.1. Client data used in WebAuthn signatures (dictionary
5481      CollectedClientData) (2)
5482
5483  #client-dataReferenced in:
5484    * 4.2. Authenticator Responses (interface AuthenticatorResponse)
5485    * 5. WebAuthn Authenticator model (2) (3) (4)
5486    * 5.1. Authenticator data (2)
5487    * 6.1. Registering a new credential
5488    * 6.2. Verifying an authentication assertion
5489    * 8. WebAuthn Extensions
5490    * 8.4. Client extension processing
5491    * 8.6. Example Extension
5492
5493  #dom-collectedclientdata-challengeReferenced in:
5494    * 4.1.3. Create a new credential - PublicKeyCredential's
5495      [[Create]](options) method
5496    * 4.1.4. Use an existing credential to make an assertion -
5497      PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5498      method
5499    * 4.7.1. Client data used in WebAuthn signatures (dictionary
5500      CollectedClientData)
5501    * 6.1. Registering a new credential
5502    * 6.2. Verifying an authentication assertion
5503
5504  #dom-collectedclientdata-originReferenced in:
5505    * 4.1.3. Create a new credential - PublicKeyCredential's
5506      [[Create]](options) method
5507    * 4.1.4. Use an existing credential to make an assertion -
5508      PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5509      method
5510    * 4.7.1. Client data used in WebAuthn signatures (dictionary
5511      CollectedClientData)
5512    * 6.1. Registering a new credential
5513    * 6.2. Verifying an authentication assertion
5514
5515  #dom-collectedclientdata-hashalgorithmReferenced in:
5516    * 4.1.3. Create a new credential - PublicKeyCredential's
5517      [[Create]](options) method
5518    * 4.1.4. Use an existing credential to make an assertion -
5519      PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5520      method
5521    * 4.7.1. Client data used in WebAuthn signatures (dictionary
5522      CollectedClientData) (2)
5523    * 6.1. Registering a new credential
5524    * 6.2. Verifying an authentication assertion
5525
5526  #dom-collectedclientdata-tokenbindingidReferenced in:
5527    * 4.1.3. Create a new credential - PublicKeyCredential's
5528      [[Create]](options) method
```

```
6739      PublicKeyCredentialRequestOptions) (2)
6740    * 5.8.1. Client data used in WebAuthn signatures (dictionary
6741      CollectedClientData) (2)
6742
6743  #dictdef-collectedclientdataReferenced in:
6744    * 5.1.3. Create a new credential - PublicKeyCredential's
6745      [[Create]](origin, options, sameOriginWithAncestors) method
6746    * 5.1.4.1. PublicKeyCredential's
6747      [[DiscoverFromExternalSource]](origin, options,
6748      sameOriginWithAncestors) method
6749    * 5.8.1. Client data used in WebAuthn signatures (dictionary
6750      CollectedClientData) (2)
6751
6752  #client-dataReferenced in:
6753    * 5.2. Authenticator Responses (interface AuthenticatorResponse)
6754    * 6. WebAuthn Authenticator model (2) (3) (4)
6755    * 6.1. Authenticator data (2)
6756    * 7.1. Registering a new credential
6757    * 7.2. Verifying an authentication assertion
6758    * 9. WebAuthn Extensions
6759    * 9.4. Client extension processing
6760    * 9.6. Example Extension
6761
6762  #dom-collectedclientdata-typeReferenced in:
6763    * 5.1.3. Create a new credential - PublicKeyCredential's
6764      [[Create]](origin, options, sameOriginWithAncestors) method
6765    * 5.1.4.1. PublicKeyCredential's
6766      [[DiscoverFromExternalSource]](origin, options,
6767      sameOriginWithAncestors) method
6768    * 5.8.1. Client data used in WebAuthn signatures (dictionary
6769      CollectedClientData)
6770    * 7.1. Registering a new credential
6771    * 7.2. Verifying an authentication assertion
6772
6773  #dom-collectedclientdata-challengeReferenced in:
6774    * 5.1.3. Create a new credential - PublicKeyCredential's
6775      [[Create]](origin, options, sameOriginWithAncestors) method
6776    * 5.1.4.1. PublicKeyCredential's
6777      [[DiscoverFromExternalSource]](origin, options,
6778      sameOriginWithAncestors) method
6779    * 5.8.1. Client data used in WebAuthn signatures (dictionary
6780      CollectedClientData)
6781    * 7.1. Registering a new credential
6782    * 7.2. Verifying an authentication assertion
6783
6784  #dom-collectedclientdata-originReferenced in:
6785    * 5.1.3. Create a new credential - PublicKeyCredential's
6786      [[Create]](origin, options, sameOriginWithAncestors) method
6787    * 5.1.4.1. PublicKeyCredential's
6788      [[DiscoverFromExternalSource]](origin, options,
6789      sameOriginWithAncestors) method
6790    * 5.8.1. Client data used in WebAuthn signatures (dictionary
6791      CollectedClientData)
6792    * 7.1. Registering a new credential
6793    * 7.2. Verifying an authentication assertion
6794
6795  #dom-collectedclientdata-hashalgorithmReferenced in:
6796    * 5.1.3. Create a new credential - PublicKeyCredential's
6797      [[Create]](origin, options, sameOriginWithAncestors) method
6798    * 5.1.4.1. PublicKeyCredential's
6799      [[DiscoverFromExternalSource]](origin, options,
6800      sameOriginWithAncestors) method
6801    * 5.8.1. Client data used in WebAuthn signatures (dictionary
6802      CollectedClientData) (2)
6803    * 7.1. Registering a new credential
6804    * 7.2. Verifying an authentication assertion
6805
6806  #dom-collectedclientdata-tokenbindingidReferenced in:
6807    * 5.1.3. Create a new credential - PublicKeyCredential's
6808      [[Create]](origin, options, sameOriginWithAncestors) method
```

```
5529      * 4.1.4. Use an existing credential to make an assertion -
5530        PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5531        method
5532      * 4.7.1. Client data used in WebAuthn signatures (dictionary
5533        CollectedClientData)
5534      * 6.1. Registering a new credential
5535      * 6.2. Verifying an authentication assertion
5536
5537    #dom-collectedclientdata-clientextensionsReferenced in:
5538      * 4.1.3. Create a new credential - PublicKeyCredential's
5539        [[Create]](options) method
5540      * 4.1.4. Use an existing credential to make an assertion -
5541        PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5542        method
5543      * 4.7.1. Client data used in WebAuthn signatures (dictionary
5544        CollectedClientData)
5545      * 6.1. Registering a new credential
5546      * 6.2. Verifying an authentication assertion
5547      * 8.4. Client extension processing
5548
5549    #dom-collectedclientdata-authenticatorextensionsReferenced in:
5550      * 4.1.3. Create a new credential - PublicKeyCredential's
5551        [[Create]](options) method
5552      * 4.1.4. Use an existing credential to make an assertion -
5553        PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5554        method
5555      * 4.7.1. Client data used in WebAuthn signatures (dictionary
5556        CollectedClientData)
5557      * 6.1. Registering a new credential
5558      * 6.2. Verifying an authentication assertion
5559
5560    #collectedclientdata-json-serialized-client-dataReferenced in:
5561      * 4.1.3. Create a new credential - PublicKeyCredential's
5562        [[Create]](options) method
5563      * 4.1.4. Use an existing credential to make an assertion -
5564        PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5565        method
5566      * 4.2. Authenticator Responses (interface AuthenticatorResponse)
5567      * 4.2.1. Information about Public Key Credential (interface
5568        AuthenticatorAttestationResponse) (2)
5569      * 4.2.2. Web Authentication Assertion (interface
5570        AuthenticatorAssertionResponse)
5571      * 4.7.1. Client data used in WebAuthn signatures (dictionary
5572        CollectedClientData)
5573
5574    #collectedclientdata-hash-of-the-serialized-client-dataReferenced in:
5575      * 4.1.3. Create a new credential - PublicKeyCredential's
5576        [[Create]](options) method (2)
5577      * 4.1.4. Use an existing credential to make an assertion -
5578        PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5579        method (2)
5580      * 4.2.1. Information about Public Key Credential (interface
5581        AuthenticatorAttestationResponse)
5582      * 4.2.2. Web Authentication Assertion (interface
5583        AuthenticatorAssertionResponse)
5584      * 4.7.1. Client data used in WebAuthn signatures (dictionary
5585        CollectedClientData)
5586      * 5. WebAuthn Authenticator model
5587      * 5.2.1. The authenticatorMakeCredential operation (2)
5588      * 5.2.2. The authenticatorGetAssertion operation (2) (3)
5589      * 5.3.2. Attestation Statement Formats (2)
5590      * 5.3.4. Generating an Attestation Object
5591      * 6.1. Registering a new credential
5592      * 7.2. Packed Attestation Statement Format (2)
5593      * 7.3. TPM Attestation Statement Format (2)
5594      * 7.4. Android Key Attestation Statement Format (2)
5595      * 7.5. Android SafetyNet Attestation Statement Format
5596      * 7.6. FIDO U2F Attestation Statement Format (2)
5597
5598    #enumdef-publickeycredentialtypeReferenced in:
```

```
6809      * 5.1.4.1. PublicKeyCredential's
6810        [[DiscoverFromExternalSource]](origin, options,
6811        sameOriginWithAncestors) method
6812      * 5.8.1. Client data used in WebAuthn signatures (dictionary
6813        CollectedClientData)
6814      * 7.1. Registering a new credential
6815      * 7.2. Verifying an authentication assertion
6816
6817    #dom-collectedclientdata-clientextensionsReferenced in:
6818      * 5.1.3. Create a new credential - PublicKeyCredential's
6819        [[Create]](origin, options, sameOriginWithAncestors) method
6820      * 5.1.4.1. PublicKeyCredential's
6821        [[DiscoverFromExternalSource]](origin, options,
6822        sameOriginWithAncestors) method
6823      * 5.8.1. Client data used in WebAuthn signatures (dictionary
6824        CollectedClientData)
6825      * 7.1. Registering a new credential
6826      * 7.2. Verifying an authentication assertion
6827      * 9.4. Client extension processing
6828
6829    #dom-collectedclientdata-authenticatorextensionsReferenced in:
6830      * 5.1.3. Create a new credential - PublicKeyCredential's
6831        [[Create]](origin, options, sameOriginWithAncestors) method
6832      * 5.1.4.1. PublicKeyCredential's
6833        [[DiscoverFromExternalSource]](origin, options,
6834        sameOriginWithAncestors) method
6835      * 5.8.1. Client data used in WebAuthn signatures (dictionary
6836        CollectedClientData)
6837      * 7.1. Registering a new credential
6838      * 7.2. Verifying an authentication assertion
6839
6840    #collectedclientdata-json-serialized-client-dataReferenced in:
6841      * 5.1.3. Create a new credential - PublicKeyCredential's
6842        [[Create]](origin, options, sameOriginWithAncestors) method
6843      * 5.1.4.1. PublicKeyCredential's
6844        [[DiscoverFromExternalSource]](origin, options,
6845        sameOriginWithAncestors) method
6846      * 5.2. Authenticator Responses (interface AuthenticatorResponse)
6847      * 5.2.1. Information about Public Key Credential (interface
6848        AuthenticatorAttestationResponse) (2)
6849      * 5.2.2. Web Authentication Assertion (interface
6850        AuthenticatorAssertionResponse)
6851      * 5.8.1. Client data used in WebAuthn signatures (dictionary
6852        CollectedClientData)
6853
6854    #collectedclientdata-hash-of-the-serialized-client-dataReferenced in:
6855      * 5.1.3. Create a new credential - PublicKeyCredential's
6856        [[Create]](origin, options, sameOriginWithAncestors) method (2)
6857      * 5.1.4.1. PublicKeyCredential's
6858        [[DiscoverFromExternalSource]](origin, options,
6859        sameOriginWithAncestors) method (2)
6860      * 5.2.1. Information about Public Key Credential (interface
6861        AuthenticatorAttestationResponse)
6862      * 5.2.2. Web Authentication Assertion (interface
6863        AuthenticatorAssertionResponse)
6864      * 5.8.1. Client data used in WebAuthn signatures (dictionary
6865        CollectedClientData)
6866      * 6. WebAuthn Authenticator model
6867      * 6.2.1. The authenticatorMakeCredential operation
6868      * 6.2.2. The authenticatorGetAssertion operation (2)
6869      * 6.3.2. Attestation Statement Formats (2)
6870      * 6.3.4. Generating an Attestation Object
6871      * 7.1. Registering a new credential
6872      * 8.2. Packed Attestation Statement Format
6873      * 8.3. TPM Attestation Statement Format
6874      * 8.4. Android Key Attestation Statement Format
6875      * 8.5. Android SafetyNet Attestation Statement Format
6876      * 8.6. FIDO U2F Attestation Statement Format
6877
6878    #enumdef-publickeycredentialtypeReferenced in:
```

```
5599      * 4.1.3. Create a new credential - PublicKeyCredential's
5600        [[Create]](options) method (2)
5601      * 4.3. Parameters for Credential Generation (dictionary
5602        PublicKeyCredentialParameters)
5603      * 4.7.2. Credential Type enumeration (enum PublicKeyCredentialType)
5604      * 4.7.3. Credential Descriptor (dictionary
5605        PublicKeyCredentialDescriptor)
5606      * 5.2.1. The authenticatorMakeCredential operation (2) (3)
5607
5608    #dom-publickeycredentialtype-public-keyReferenced in:
5609      * 4.7.2. Credential Type enumeration (enum PublicKeyCredentialType)
5610
5611    #dictdef-publickeycredentialdescriptorReferenced in:
5612      * 4.4. Options for Credential Creation (dictionary
5613        MakePublicKeyCredentialOptions) (2)
5614      * 4.5. Options for Assertion Generation (dictionary
5615        PublicKeyCredentialRequestOptions) (2) (3)
5616      * 4.7.3. Credential Descriptor (dictionary
5617        PublicKeyCredentialDescriptor)
5618      * 5.2.1. The authenticatorMakeCredential operation
5619
5620    #dom-publickeycredentialdescriptor-transportsReferenced in:
5621      * 4.1.3. Create a new credential - PublicKeyCredential's
5622        [[Create]](options) method (2)
5623      * 4.1.4. Use an existing credential to make an assertion -
5624        PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5625        method (2)
5626
5627    #dom-publickeycredentialdescriptor-typeReferenced in:
5628      * 4.1.4. Use an existing credential to make an assertion -
5629        PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5630        method
5631      * 4.7.3. Credential Descriptor (dictionary
5632        PublicKeyCredentialDescriptor)
5633
5634    #dom-publickeycredentialdescriptor-idReferenced in:
5635      * 4.1.4. Use an existing credential to make an assertion -
5636        PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5637        method
5638      * 4.7.3. Credential Descriptor (dictionary
5639        PublicKeyCredentialDescriptor)
5640
5641    #enumdef-authenticatortransportReferenced in:
5642      * 4.7.3. Credential Descriptor (dictionary
5643        PublicKeyCredentialDescriptor)
5644      * 4.7.4. Authenticator Transport enumeration (enum
5645        AuthenticatorTransport)
5646
5647    #dom-authenticatortransport-usbReferenced in:
5648      * 4.7.4. Authenticator Transport enumeration (enum
5649        AuthenticatorTransport)
5650
5651    #dom-authenticatortransport-nfcReferenced in:
5652      * 4.7.4. Authenticator Transport enumeration (enum
5653        AuthenticatorTransport)
5654
5655    #dom-authenticatortransport-bleReferenced in:
5656      * 4.7.4. Authenticator Transport enumeration (enum
5657        AuthenticatorTransport)
5658
5659    #typedefdef-cosealgorithmidentifierReferenced in:
5660      * 4.1.3. Create a new credential - PublicKeyCredential's
```

```
6879      * 5.1.3. Create a new credential - PublicKeyCredential's
6880        [[Create]](origin, options, sameOriginWithAncestors) method (2)
6881      * 5.3. Parameters for Credential Generation (dictionary
6882        PublicKeyCredentialParameters)
6883      * 5.8.2. Credential Type enumeration (enum PublicKeyCredentialType)
6884      * 5.8.3. Credential Descriptor (dictionary
6885        PublicKeyCredentialDescriptor)
6886      * 6.2.1. The authenticatorMakeCredential operation (2) (3)
6887
6888    #dom-publickeycredentialtype-public-keyReferenced in:
6889      * 5.8.2. Credential Type enumeration (enum PublicKeyCredentialType)
6890
6891    #dictdef-publickeycredentialdescriptorReferenced in:
6892      * 5.1.4.1. PublicKeyCredential's
6893        [[DiscoverFromExternalSource]](origin, options,
6894        sameOriginWithAncestors) method
6895      * 5.4. Options for Credential Creation (dictionary
6896        MakePublicKeyCredentialOptions) (2)
6897      * 5.5. Options for Assertion Generation (dictionary
6898        PublicKeyCredentialRequestOptions) (2) (3)
6899      * 5.8.3. Credential Descriptor (dictionary
6900        PublicKeyCredentialDescriptor)
6901      * 6.2.1. The authenticatorMakeCredential operation
6902      * 6.2.2. The authenticatorGetAssertion operation
6903
6904    #dom-publickeycredentialdescriptor-transportsReferenced in:
6905      * 5.1.3. Create a new credential - PublicKeyCredential's
6906        [[Create]](origin, options, sameOriginWithAncestors) method (2)
6907      * 5.1.4.1. PublicKeyCredential's
6908        [[DiscoverFromExternalSource]](origin, options,
6909        sameOriginWithAncestors) method (2)
6910
6911    #dom-publickeycredentialdescriptor-typeReferenced in:
6912      * 5.1.4.1. PublicKeyCredential's
6913        [[DiscoverFromExternalSource]](origin, options,
6914        sameOriginWithAncestors) method
6915      * 5.8.3. Credential Descriptor (dictionary
6916        PublicKeyCredentialDescriptor)
6917      * 6.2.1. The authenticatorMakeCredential operation
6918      * 6.2.2. The authenticatorGetAssertion operation
6919
6920    #dom-publickeycredentialdescriptor-idReferenced in:
6921      * 5.1.4.1. PublicKeyCredential's
6922        [[DiscoverFromExternalSource]](origin, options,
6923        sameOriginWithAncestors) method (2)
6924      * 5.8.3. Credential Descriptor (dictionary
6925        PublicKeyCredentialDescriptor)
6926      * 6.2.1. The authenticatorMakeCredential operation
6927      * 6.2.2. The authenticatorGetAssertion operation
6928
6929    #enumdef-authenticatortransportReferenced in:
6930      * 5.8.3. Credential Descriptor (dictionary
6931        PublicKeyCredentialDescriptor)
6932      * 5.8.4. Authenticator Transport enumeration (enum
6933        AuthenticatorTransport)
6934
6935    #dom-authenticatortransport-usbReferenced in:
6936      * 5.8.4. Authenticator Transport enumeration (enum
6937        AuthenticatorTransport)
6938
6939    #dom-authenticatortransport-nfcReferenced in:
6940      * 5.8.4. Authenticator Transport enumeration (enum
6941        AuthenticatorTransport)
6942
6943    #dom-authenticatortransport-bleReferenced in:
6944      * 5.8.4. Authenticator Transport enumeration (enum
6945        AuthenticatorTransport)
6946
6947    #typedefdef-cosealgorithmidentifierReferenced in:
6948      * 5.1.3. Create a new credential - PublicKeyCredential's
```

```
[[Create]](options) method
* 4.3. Parameters for Credential Generation (dictionary
PublicKeyCredentialParameters)
* 4.7.5. Cryptographic Algorithm Identifier (typedef
COSEAlgorithmIdentifier)
* 5.2.1. The authenticatorMakeCredential operation
* 5.3.1. Attestation data
```

```
#attestation-signatureReferenced in:
  * 3. Terminology
  * 5. WebAuthn Authenticator model (2) (3)
  * 5.3. Attestation
  * 7.6. FIDO U2F Attestation Statement Format

#assertion-signatureReferenced in:
  * 5. WebAuthn Authenticator model (2)
  * 5.2.2. The authenticatorGetAssertion operation (2) (3) (4) (5) (6)

#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. Attestation (2)
  * 5.3.1. Attestation data
```

```
[[Create]](origin, options, sameOriginWithAncestors) method
* 5.3. Parameters for Credential Generation (dictionary
PublicKeyCredentialParameters)
* 5.8.5. Cryptographic Algorithm Identifier (typedef
COSEAlgorithmIdentifier)
* 6.2.1. The authenticatorMakeCredential operation
* 6.3.1. Attested credential data
* 8.2. Packed Attestation Statement Format
* 8.3. TPM Attestation Statement Format

#enumdef-userverificationrequirementReferenced in:
  * 5.4.4. Authenticator Selection Criteria (dictionary
  AuthenticatorSelectionCriteria) (2)
  * 5.5. Options for Assertion Generation (dictionary
  PublicKeyCredentialRequestOptions) (2)
  * 5.8.6. User Verification Requirement enumeration (enum
  UserVerificationRequirement)

#dom-userverificationrequirement-requiredReferenced in:
  * 5.1.3. Create a new credential - PublicKeyCredential's
  [[Create]](origin, options, sameOriginWithAncestors) method (2)
  * 5.1.4.1. PublicKeyCredential's
  [[DiscoverFromExternalSource]](origin, options,
  sameOriginWithAncestors) method (2)
  * 5.8.6. User Verification Requirement enumeration (enum
  UserVerificationRequirement)

#dom-userverificationrequirement-preferredReferenced in:
  * 5.1.3. Create a new credential - PublicKeyCredential's
  [[Create]](origin, options, sameOriginWithAncestors) method
  * 5.1.4.1. PublicKeyCredential's
  [[DiscoverFromExternalSource]](origin, options,
  sameOriginWithAncestors) method
  * 5.8.6. User Verification Requirement enumeration (enum
  UserVerificationRequirement)

#dom-userverificationrequirement-discouragedReferenced in:
  * 5.1.3. Create a new credential - PublicKeyCredential's
  [[Create]](origin, options, sameOriginWithAncestors) method
  * 5.1.4.1. PublicKeyCredential's
  [[DiscoverFromExternalSource]](origin, options,
  sameOriginWithAncestors) method
  * 5.8.6. User Verification Requirement enumeration (enum
  UserVerificationRequirement)

#attestation-signatureReferenced in:
  * 4. Terminology
  * 6. WebAuthn Authenticator model (2) (3)
  * 6.3. Attestation
  * 7.1. Registering a new credential
  * 8.6. FIDO U2F Attestation Statement Format

#assertion-signatureReferenced in:
  * 6. WebAuthn Authenticator model (2)
  * 6.2.2. The authenticatorGetAssertion operation (2) (3)

#authenticator-dataReferenced in:
  * 5.1.4.1. PublicKeyCredential's
  [[DiscoverFromExternalSource]](origin, options,
  sameOriginWithAncestors) method
  * 5.2.1. Information about Public Key Credential (interface
AuthenticatorAttestationResponse) (2)
  * 5.2.2. Web Authentication Assertion (interface
AuthenticatorAssertionResponse)
  * 6. WebAuthn Authenticator model (2)
  * 6.1. Authenticator data (2) (3) (4) (5) (6) (7) (8) (9)
  * 6.1.1. Signature Counter Considerations (2)
  * 6.2.1. The authenticatorMakeCredential operation
  * 6.2.2. The authenticatorGetAssertion operation
  * 6.3. Attestation (2)
```

```
5690        * 5.3.2. Attestation Statement Formats (2)
5691        * 5.3.4. Generating an Attestation Object (2) (3)
5692        * 5.3.5.3. Attestation Certificate Hierarchy
5693        * 6.1. Registering a new credential (2)
5694        * 7.5. Android SafetyNet Attestation Statement Format
5695        * 8.5. Authenticator extension processing (2)
5696        * 8.6. Example Extension (2)
5697        * 9.6. User Verification Index Extension (uvi)
5698        * 9.7. Location Extension (loc)
5699        * 9.8. User Verification Method Extension (uvm)
5700
5701    #authenticatormakecredentialReferenced in:
5702      * 3. Terminology (2) (3)
5703      * 4.1.3. Create a new credential - PublicKeyCredential's
5704        [[Create]](options) method (2)
5705      * 5. WebAuthn Authenticator model
5706      * 5.2.3. The authenticatorCancel operation (2)
5707      * 8. WebAuthn Extensions
5708      * 8.2. Defining extensions
5709
5710    #authenticatorgetassertionReferenced in:
5711      * 3. Terminology (2) (3)
5712      * 4.1.4. Use an existing credential to make an assertion -
5713        PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5714        method (2) (3) (4)
5715      * 5. WebAuthn Authenticator model
5716      * 5.1. Authenticator data
```

```
7019        * 6.3.1. Attested credential data
7020        * 6.3.2. Attestation Statement Formats (2)
7021        * 6.3.4. Generating an Attestation Object
7022        * 6.3.5.3. Attestation Certificate Hierarchy
7023        * 7.1. Registering a new credential
7024        * 8.5. Android SafetyNet Attestation Statement Format
7025        * 9.5. Authenticator extension processing
7026        * 9.6. Example Extension (2)
7027        * 10.6. User Verification Index Extension (uvi)
7028        * 10.7. Location Extension (loc)
7029        * 10.8. User Verification Method Extension (uvm)
7030
7031    #rpidhashReferenced in:
7032      * 7.2. Verifying an authentication assertion
7033
7034    #flagsReferenced in:
7035      * 5.8.6. User Verification Requirement enumeration (enum
7036        UserVerificationRequirement) (2)
7037      * 6.1. Authenticator data
7038
7039    #signcountReferenced in:
7040      * 6.1.1. Signature Counter Considerations (2)
7041      * 7.2. Verifying an authentication assertion (2) (3)
7042
7043    #attestedcredentialdataReferenced in:
7044      * 5.1.3. Create a new credential - PublicKeyCredential's
7045        [[Create]](origin, options, sameOriginWithAncestors) method
7046      * 6.1. Authenticator data (2)
7047      * 6.2.1. The authenticatorMakeCredential operation
7048      * 6.2.2. The authenticatorGetAssertion operation
7049      * 7.1. Registering a new credential (2)
7050      * 8.3. TPM Attestation Statement Format
7051      * 8.4. Android Key Attestation Statement Format
7052      * 8.6. FIDO U2F Attestation Statement Format
7053
7054    #authdataextensionsReferenced in:
7055      * 6.1. Authenticator data
7056      * 6.2.1. The authenticatorMakeCredential operation
7057      * 6.2.2. The authenticatorGetAssertion operation
7058
7059    #signature-counterReferenced in:
7060      * 6.1. Authenticator data
7061      * 6.1.1. Signature Counter Considerations (2) (3) (4) (5) (6) (7) (8)
7062        (9) (10)
7063      * 6.2.1. The authenticatorMakeCredential operation (2) (3) (4)
7064      * 6.2.2. The authenticatorGetAssertion operation (2)
7065      * 7.2. Verifying an authentication assertion (2) (3) (4) (5) (6)
7066
7067    #authenticator-sessionReferenced in:
7068      * 5.6. Abort operations with AbortSignal (2)
7069      * 6.2.1. The authenticatorMakeCredential operation
7070      * 6.2.2. The authenticatorGetAssertion operation
7071      * 6.2.3. The authenticatorCancel operation (2)
7072
7073    #authenticatormakecredentialReferenced in:
7074      * 4. Terminology (2) (3) (4)
7075      * 5.1.3. Create a new credential - PublicKeyCredential's
7076        [[Create]](origin, options, sameOriginWithAncestors) method (2)
7077      * 6. WebAuthn Authenticator model
7078      * 6.2.3. The authenticatorCancel operation (2)
7079      * 9. WebAuthn Extensions
7080      * 9.2. Defining extensions
7081
7082    #authenticatorgetassertionReferenced in:
7083      * 4. Terminology (2) (3)
7084      * 5.1.4.1. PublicKeyCredential's
7085        [[DiscoverFromExternalSource]](origin, options,
7086        sameOriginWithAncestors) method (2) (3) (4)
7087      * 6. WebAuthn Authenticator model
7088      * 6.1. Authenticator data
```

* 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 to make an assertion - PublicKeyCredential's [[DiscoverFromExternalSource]](options) method (2) (3)

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

#attestation-statementReferenced in:
* 3. Terminology
* 4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse) (2) (3)
* 5.3. Attestation (2) (3) (4) (5) (6) (7) (8)
* 5.3.2. Attestation Statement Formats (2) (3)

#attestation-statement-formatReferenced in:
* 4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)
* 4.7.4. Authenticator Transport enumeration (enum AuthenticatorTransport)
* 5.3. Attestation (2) (3) (4) (5) (6) (7)
* 5.3.2. Attestation Statement Formats (2) (3) (4)
* 5.3.4. Generating an Attestation Object (2)

#attestation-typeReferenced in:
* 5.3. Attestation (2) (3) (4) (5) (6)
* 5.3.2. Attestation Statement Formats

#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. 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

* 6.1.1. Signature Counter Considerations (2) (3)
* 6.2.3. The authenticatorCancel operation (2)
* 9. WebAuthn Extensions
* 9.2. Defining extensions

#authenticatorcancelReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2) (3) (4)
* 5.1.4.1. PublicKeyCredential's [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors) method (2) (3) (4)
* 6.2.1. The authenticatorMakeCredential operation
* 6.2.2. The authenticatorGetAssertion operation

#attestation-objectReferenced in:
* 4. Terminology (2) (3)
* 5. Web Authentication API
* 5.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse) (2)
* 5.4. Options for Credential Creation (dictionary MakePublicKeyCredentialOptions) (2)
* 6.2.1. The authenticatorMakeCredential operation (2)
* 6.3. Attestation (2) (3)
* 6.3.1. Attested credential data
* 6.3.4. Generating an Attestation Object (2)
* 7.1. Registering a new credential

#attestation-statementReferenced in:
* 4. Terminology (2)
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)
* 5.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse) (2) (3)
* 5.4.6. Attestation Conveyance Preference enumeration (enum AttestationConveyancePreference) (2) (3) (4) (5) (6) (7)
* 6.3. Attestation (2) (3) (4) (5) (6) (7) (8)
* 6.3.2. Attestation Statement Formats (2) (3) (4)
* 7.1. Registering a new credential

#attestation-statement-formatReferenced in:
* 5.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)
* 5.8.4. Authenticator Transport enumeration (enum AuthenticatorTransport)
* 6.2.1. The authenticatorMakeCredential operation
* 6.3. Attestation (2) (3) (4) (5) (6) (7)
* 6.3.2. Attestation Statement Formats (2) (3) (4)
* 6.3.4. Generating an Attestation Object
* 7.1. Registering a new credential (2)

#attestation-typeReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
* 6.3. Attestation (2) (3) (4) (5) (6)
* 6.3.2. Attestation Statement Formats (2)

#attested-credential-dataReferenced in:
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method
* 6.1. Authenticator data (2) (3) (4) (5)
* 6.2.1. The authenticatorMakeCredential operation
* 6.3. Attestation (2)
* 6.3.1. Attested credential data
* 6.3.3. Attestation Types

#aaguidReferenced in:
* 4. Terminology
* 5.1.3. Create a new credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) method (2) (3)

```
5770
5771    #signing-procedureReferenced in:
5772      * 5.3.2. Attestation Statement Formats
5773
5774    #authenticator-data-for-the-attestationReferenced in:
5775      * 7.2. Packed Attestation Statement Format
5776      * 7.3. TPM Attestation Statement Format
5777      * 7.4. Android Key Attestation Statement Format (2)
5778      * 7.5. Android SafetyNet Attestation Statement Format
5779      * 7.6. FIDO U2F Attestation Statement Format
5780
5781    #authenticator-data-claimed-to-have-been-used-for-the-attestationRefere
5782    nced in:
5783      * 7.2. Packed Attestation Statement Format
5784      * 7.3. TPM Attestation Statement Format
5785      * 7.4. Android Key Attestation Statement Format (2)
5786      * 7.6. FIDO U2F Attestation Statement Format
5787
5788    #basic-attestationReferenced in:
5789      * 5.3.5.1. Privacy
5790
5791    #self-attestationReferenced in:
5792      * 3. Terminology (2) (3) (4)
5793      * 5.3. Attestation (2)
5794      * 5.3.2. Attestation Statement Formats
5795      * 5.3.3. Attestation Types
5796      * 5.3.5.2. Attestation Certificate and Attestation Certificate CA
5797        Compromise
5798      * 6.1. Registering a new credential (2) (3)
5799      * 7.2. Packed Attestation Statement Format (2)
5800      * 7.6. FIDO U2F Attestation Statement Format
```

```
7159      * 5.4.6. Attestation Conveyance Preference enumeration (enum
7160        AttestationConveyancePreference)
7161      * 7.1. Registering a new credential
7162      * 8.2. Packed Attestation Statement Format
7163      * 8.3. TPM Attestation Statement Format
7164
7165    #credentialidlengthReferenced in:
7166      * 6.1. Authenticator data
7167
7168    #credentialidReferenced in:
7169      * 5.1.3. Create a new credential - PublicKeyCredential's
7170      [[Create]](origin, options, sameOriginWithAncestors) method
7171      * 6.1. Authenticator data
7172      * 7.1. Registering a new credential
7173
7174    #credentialpublickeyReferenced in:
7175      * 6.1. Authenticator data
7176      * 7.1. Registering a new credential
7177      * 8.2. Packed Attestation Statement Format
7178      * 8.3. TPM Attestation Statement Format
7179      * 8.4. Android Key Attestation Statement Format
7180
7181    #signing-procedureReferenced in:
7182      * 6.3.2. Attestation Statement Formats
7183      * 6.3.4. Generating an Attestation Object
7184
7185    #authenticator-data-for-the-attestationReferenced in:
7186      * 8.2. Packed Attestation Statement Format
7187      * 8.3. TPM Attestation Statement Format
7188      * 8.4. Android Key Attestation Statement Format (2)
7189      * 8.5. Android SafetyNet Attestation Statement Format
7190      * 8.6. FIDO U2F Attestation Statement Format
7191
7192    #verification-procedure-inputsReferenced in:
7193      * 8.2. Packed Attestation Statement Format
7194      * 8.3. TPM Attestation Statement Format
7195      * 8.4. Android Key Attestation Statement Format
7196      * 8.5. Android SafetyNet Attestation Statement Format
7197      * 8.6. FIDO U2F Attestation Statement Format
7198
7199    #authenticator-data-claimed-to-have-been-used-for-the-attestationRefere
7200    nced in:
7201      * 8.4. Android Key Attestation Statement Format
7202
7203    #attestation-trust-pathReferenced in:
7204      * 6.3.2. Attestation Statement Formats
7205      * 8.2. Packed Attestation Statement Format (2) (3)
7206      * 8.3. TPM Attestation Statement Format
7207      * 8.4. Android Key Attestation Statement Format
7208      * 8.5. Android SafetyNet Attestation Statement Format
7209      * 8.6. FIDO U2F Attestation Statement Format
7210
7211    #basic-attestationReferenced in:
7212      * 6.3.5.1. Privacy
7213      * 8.4. Android Key Attestation Statement Format
7214      * 8.5. Android SafetyNet Attestation Statement Format
7215      * 8.6. FIDO U2F Attestation Statement Format
7216
7217    #self-attestationReferenced in:
7218      * 4. Terminology (2) (3) (4)
7219      * 5.4.6. Attestation Conveyance Preference enumeration (enum
7220        AttestationConveyancePreference)
7221      * 6.3. Attestation (2)
7222      * 6.3.2. Attestation Statement Formats
7223      * 6.3.3. Attestation Types
7224      * 6.3.5.2. Attestation Certificate and Attestation Certificate CA
7225        Compromise
7226      * 7.1. Registering a new credential (2) (3)
7227      * 8.2. Packed Attestation Statement Format (2)
7228      * 8.6. FIDO U2F Attestation Statement Format
```

```
#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
  * 9.2. Simple Transaction Authorization Extension (txAuthSimple)
  * 9.3. Generic Transaction Authorization Extension (txAuthGeneric)
  * 9.4. Authenticator Selection Extension (authnSel)
  * 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)
  * 10.2. WebAuthn Extension Identifier Registrations (2) (3) (4) (5)
    (6) (7)

#authentication-extensionReferenced in:
  * 4.1.4. Use an existing credential to make an assertion -
    PublicKeyCredential's [[DiscoverFromExternalSource]](options)
    method
  * 8. WebAuthn Extensions (2) (3) (4) (5) (6)
  * 8.6. Example Extension
  * 9.1. FIDO AppId Extension (appid)
  * 9.2. Simple Transaction Authorization Extension (txAuthSimple)
  * 9.3. Generic Transaction Authorization Extension (txAuthGeneric)
  * 9.6. User Verification Index Extension (uvi)
  * 9.7. Location Extension (loc)
  * 9.8. User Verification Method Extension (uvm)
  * 10.2. WebAuthn Extension Identifier Registrations (2) (3) (4) (5)
    (6)

#client-extensionReferenced in:
  * 4.1.3. Create a new credential - PublicKeyCredential's
    [[Create]](options) method
```

```
#privacy-caReferenced in:
  * 5.1.3. Create a new credential - PublicKeyCredential's
    [[Create]](origin, options, sameOriginWithAncestors) method
  * 5.4.6. Attestation Conveyance Preference enumeration (enum
    AttestationConveyancePreference)
  * 6.3.5.1. Privacy
  * 8.3. TPM Attestation Statement Format
  * 8.6. FIDO U2F Attestation Statement Format

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

#ecdaaReferenced in:
  * 6.3.2. Attestation Statement Formats
  * 6.3.3. Attestation Types
  * 6.3.5.2. Attestation Certificate and Attestation Certificate CA
    Compromise
  * 7.1. Registering a new credential
  * 8.2. Packed Attestation Statement Format (2)
  * 8.3. TPM Attestation Statement Format (2) (3)

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

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

#ecdaa-issuer-public-keyReferenced in:
  * 6.3.2. Attestation Statement Formats
  * 6.3.5.1. Privacy
  * 7.1. Registering a new credential
  * 8.2. Packed Attestation Statement Format (2) (3)

#registration-extensionReferenced in:
  * 5.1.3. Create a new credential - PublicKeyCredential's
    [[Create]](origin, options, sameOriginWithAncestors) method
  * 9. WebAuthn Extensions (2) (3) (4) (5) (6)
  * 9.6. Example Extension
  * 10.2. Simple Transaction Authorization Extension (txAuthSimple)
  * 10.3. Generic Transaction Authorization Extension (txAuthGeneric)
  * 10.4. Authenticator Selection Extension (authnSel)
  * 10.5. Supported Extensions Extension (exts)
  * 10.6. User Verification Index Extension (uvi)
  * 10.7. Location Extension (loc)
  * 10.8. User Verification Method Extension (uvm)
  * 11.2. WebAuthn Extension Identifier Registrations (2) (3) (4) (5)
    (6) (7)

#authentication-extensionReferenced in:
  * 5.1.4.1. PublicKeyCredential's
    [[DiscoverFromExternalSource]](origin, options,
    sameOriginWithAncestors) method
  * 9. WebAuthn Extensions (2) (3) (4) (5) (6)
  * 9.6. Example Extension
  * 10.1. FIDO AppId Extension (appid)
  * 10.2. Simple Transaction Authorization Extension (txAuthSimple)
  * 10.3. Generic Transaction Authorization Extension (txAuthGeneric)
  * 10.6. User Verification Index Extension (uvi)
  * 10.7. Location Extension (loc)
  * 10.8. User Verification Method Extension (uvm)
  * 11.2. WebAuthn Extension Identifier Registrations (2) (3) (4) (5)
    (6)

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

```
5865      * 4.1.4. Use an existing credential to make an assertion -
5866        PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5867        method
5868      * 4.6. Authentication Extensions (typedef AuthenticationExtensions)
5869      * 8. WebAuthn Extensions
5870      * 8.2. Defining extensions
5871      * 8.4. Client extension processing
5872
5873    #authenticator-extensionReferenced in:
5874      * 4.1.3. Create a new credential - PublicKeyCredential's
5875        [[Create]](options) method
5876      * 4.1.4. Use an existing credential to make an assertion -
5877        PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5878        method
5879      * 4.6. Authentication Extensions (typedef AuthenticationExtensions)
5880      * 8. WebAuthn Extensions (2) (3)
5881      * 8.2. Defining extensions (2)
5882      * 8.3. Extending request parameters
5883      * 8.5. Authenticator extension processing
5884
5885    #extension-identifierReferenced in:
5886      * 4.1. PublicKeyCredential Interface
5887      * 4.1.3. Create a new credential - PublicKeyCredential's
5888        [[Create]](options) method
5889      * 4.1.4. Use an existing credential to make an assertion -
5890        PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5891        method
5892      * 5.1. Authenticator data
5893      * 8. WebAuthn Extensions (2)
5894      * 8.2. Defining extensions
5895      * 8.3. Extending request parameters
5896      * 8.4. Client extension processing (2)
5897      * 8.5. Authenticator extension processing (2)
5898      * 8.6. Example Extension
5899      * 9.5. Supported Extensions Extension (exts) (2)
5900      * 9.7. Location Extension (loc)
5901      * 10.2. WebAuthn Extension Identifier Registrations
5902
5903    #client-extension-inputReferenced in:
5904      * 8. WebAuthn Extensions (2) (3)
5905      * 8.2. Defining extensions
5906      * 8.3. Extending request parameters (2) (3) (4) (5) (6)
5907      * 8.4. Client extension processing (2) (3) (4)
5908      * 8.6. Example Extension
5909
5910    #authenticator-extension-inputReferenced in:
5911      * 8. WebAuthn Extensions (2) (3) (4) (5)
5912      * 8.2. Defining extensions
5913      * 8.3. Extending request parameters (2) (3)
5914      * 8.4. Client extension processing
5915      * 8.5. Authenticator extension processing (2) (3)
5916
5917    #client-extension-processingReferenced in:
5918      * 4.1. PublicKeyCredential Interface
5919      * 4.1.3. Create a new credential - PublicKeyCredential's
5920        [[Create]](options) method (2)
5921      * 4.1.4. Use an existing credential to make an assertion -
5922        PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5923        method (2)
5924      * 8. WebAuthn Extensions (2) (3) (4)
5925      * 8.2. Defining extensions
5926
5927    #client-extension-outputReferenced in:
5928      * 4.1. PublicKeyCredential Interface
5929      * 4.1.3. Create a new credential - PublicKeyCredential's
5930        [[Create]](options) method (2)
```

```
7299      * 5.1.4.1. PublicKeyCredential's
7300        [[DiscoverFromExternalSource]](origin, options,
7301        sameOriginWithAncestors) method
7302      * 5.7. Authentication Extensions (typedef AuthenticationExtensions)
7303      * 9. WebAuthn Extensions
7304      * 9.2. Defining extensions
7305      * 9.4. Client extension processing
7306
7307    #authenticator-extensionReferenced in:
7308      * 5.1.3. Create a new credential - PublicKeyCredential's
7309        [[Create]](origin, options, sameOriginWithAncestors) method
7310      * 5.1.4.1. PublicKeyCredential's
7311        [[DiscoverFromExternalSource]](origin, options,
7312        sameOriginWithAncestors) method
7313      * 5.7. Authentication Extensions (typedef AuthenticationExtensions)
7314      * 9. WebAuthn Extensions (2) (3)
7315      * 9.2. Defining extensions (2)
7316      * 9.3. Extending request parameters
7317      * 9.5. Authenticator extension processing
7318
7319    #extension-identifierReferenced in:
7320      * 5.1. PublicKeyCredential Interface
7321      * 5.1.3. Create a new credential - PublicKeyCredential's
7322        [[Create]](origin, options, sameOriginWithAncestors) method
7323      * 5.1.4.1. PublicKeyCredential's
7324        [[DiscoverFromExternalSource]](origin, options,
7325        sameOriginWithAncestors) method
7326      * 6.1. Authenticator data
7327      * 6.2.1. The authenticatorMakeCredential operation (2)
7328      * 6.2.2. The authenticatorGetAssertion operation (2)
7329      * 9. WebAuthn Extensions (2)
7330      * 9.2. Defining extensions
7331      * 9.3. Extending request parameters
7332      * 9.4. Client extension processing (2)
7333      * 9.5. Authenticator extension processing (2)
7334      * 9.6. Example Extension
7335      * 10.5. Supported Extensions Extension (exts) (2)
7336      * 10.7. Location Extension (loc)
7337      * 11.2. WebAuthn Extension Identifier Registrations
7338
7339    #client-extension-inputReferenced in:
7340      * 9. WebAuthn Extensions (2) (3)
7341      * 9.2. Defining extensions
7342      * 9.3. Extending request parameters (2) (3) (4) (5) (6)
7343      * 9.4. Client extension processing (2) (3) (4)
7344      * 9.6. Example Extension
7345
7346    #authenticator-extension-inputReferenced in:
7347      * 6.2.1. The authenticatorMakeCredential operation
7348      * 6.2.2. The authenticatorGetAssertion operation
7349      * 9. WebAuthn Extensions (2) (3) (4) (5)
7350      * 9.2. Defining extensions
7351      * 9.3. Extending request parameters (2) (3)
7352      * 9.4. Client extension processing
7353      * 9.5. Authenticator extension processing (2) (3)
7354
7355    #client-extension-processingReferenced in:
7356      * 5.1. PublicKeyCredential Interface
7357      * 5.1.3. Create a new credential - PublicKeyCredential's
7358        [[Create]](origin, options, sameOriginWithAncestors) method (2)
7359      * 5.1.4.1. PublicKeyCredential's
7360        [[DiscoverFromExternalSource]](origin, options,
7361        sameOriginWithAncestors) method (2)
7362      * 9. WebAuthn Extensions (2) (3) (4)
7363      * 9.2. Defining extensions
7364
7365    #client-extension-outputReferenced in:
7366      * 5.1. PublicKeyCredential Interface
7367      * 5.1.3. Create a new credential - PublicKeyCredential's
7368        [[Create]](origin, options, sameOriginWithAncestors) method (2)
```

```
5931        * 4.1.4. Use an existing credential to make an assertion -
5932          PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5933          method (2)
5934        * 8. WebAuthn Extensions (2) (3)
5935        * 8.2. Defining extensions (2) (3)
5936        * 8.4. Client extension processing (2) (3)
5937        * 8.6. Example Extension
5938
5939      #authenticator-extension-processingReferenced in:
5940        * 8. WebAuthn Extensions
5941        * 8.2. Defining extensions
5942        * 8.5. Authenticator extension processing
5943
5944      #authenticator-extension-outputReferenced in:
5945        * 5.1. Authenticator data
5946        * 8. WebAuthn Extensions (2) (3)
5947        * 8.2. Defining extensions (2) (3)
5948        * 8.4. Client extension processing
5949        * 8.5. Authenticator extension processing
5950        * 8.6. Example Extension
5951        * 9.5. Supported Extensions Extension (exts)
5952        * 9.6. User Verification Index Extension (uvi)
5953        * 9.7. Location Extension (loc)
5954        * 9.8. User Verification Method Extension (uvm)
5955
5956      #typedefdef-authenticatorselectionlistReferenced in:
5957        * 9.4. Authenticator Selection Extension (authnSel)
5958
5959      #typedefdef-aaguidReferenced in:
5960        * 9.4. Authenticator Selection Extension (authnSel)
5961
```

```
7369        * 5.1.4.1. PublicKeyCredential's
7370          [[DiscoverFromExternalSource]](origin, options,
7371          sameOriginWithAncestors) method (2)
7372        * 9. WebAuthn Extensions (2) (3)
7373        * 9.2. Defining extensions (2) (3)
7374        * 9.4. Client extension processing (2) (3)
7375        * 9.6. Example Extension
7376
7377      #authenticator-extension-processingReferenced in:
7378        * 6.2.1. The authenticatorMakeCredential operation
7379        * 6.2.2. The authenticatorGetAssertion operation
7380        * 9. WebAuthn Extensions
7381        * 9.2. Defining extensions
7382        * 9.5. Authenticator extension processing
7383
7384      #authenticator-extension-outputReferenced in:
7385        * 6.1. Authenticator data
7386        * 9. WebAuthn Extensions (2) (3)
7387        * 9.2. Defining extensions (2) (3)
7388        * 9.4. Client extension processing
7389        * 9.5. Authenticator extension processing
7390        * 9.6. Example Extension
7391        * 10.5. Supported Extensions Extension (exts)
7392        * 10.6. User Verification Index Extension (uvi)
7393        * 10.7. Location Extension (loc)
7394        * 10.8. User Verification Method Extension (uvm)
7395
7396      #typedefdef-authenticatorselectionlistReferenced in:
7397        * 10.4. Authenticator Selection Extension (authnSel)
7398
7399      #typedefdef-aaguidReferenced in:
7400        * 10.4. Authenticator Selection Extension (authnSel)
7401
```