THE_URL:file://localhost/Users/jehodges/documents/work/standards/W3C/WebAuthn/index-vgb-u2f-attestation-0d0fcea.html
THE_TITLE:Web Authentication: An API for accessing Scoped Credentials
W3C

Web Authentication: An API for accessing Scoped Credentials

Editor's Draft, 12 January 2017

This version:
https://w3c.github.io/webauthn/

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

Previous Versions:
https://www.w3.org/TR/2016/WD-webauthn-20161207/
https://www.w3.org/TR/2016/WD-webauthn-20160928/
https://www.w3.org/TR/2016/WD-webauthn-20160902/
https://www.w3.org/TR/2016/WD-webauthn-20160531/

Issue Tracking:
Github

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

Abstract

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

Status of this document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

THE_URL:file://localhost/Users/jehodges/documents/work/standards/W3C/WebAuthn/index-vgb-u2f-attestation-dc90eab.html
THE_TITLE:Web Authentication: An API for accessing Scoped Credentials
W3C

Web Authentication: An API for accessing Scoped Credentials

Editor's Draft, 18 January 2017

This version:
https://w3c.github.io/webauthn/

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

Previous Versions:
https://www.w3.org/TR/2016/WD-webauthn-20161207/
https://www.w3.org/TR/2016/WD-webauthn-20160928/
https://www.w3.org/TR/2016/WD-webauthn-20160902/
https://www.w3.org/TR/2016/WD-webauthn-20160531/

Issue Tracking:
Github

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

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

Abstract

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

Status of this document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

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

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

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

Table of Contents

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

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

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

Table of Contents

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 WebAuthentication Interface
        1. 4.1.1 Create a new credential (makeCredential() method)
        2. 4.1.2 Use an existing credential (getAssertion() method)
    2. 4.2 Information about Scoped Credential (interface ScopedCredentialInfo)
    3. 4.3 User Account Information (dictionary Account)
    4. 4.4 Parameters for Credential Generation (dictionary ScopedCredentialParameters)
    5. 4.5 Additional options for Credential Generation (dictionary ScopedCredentialOptions)
        1. 4.5.1 Credential Attachment enumeration (enum Attachment)
    6. 4.6 Web Authentication Assertion (interface AuthenticationAssertion)
    7. 4.7 Additional options for Assertion Generation (dictionary AssertionOptions)
    8. 4.8 Authentication Assertion Extensions (dictionary AuthenticationExtensions)
    9. 4.9 Supporting Data Structures
        1. 4.9.1 Client data used in WebAuthn signatures (dictionary ClientData)
        2. 4.9.2 Credential Type enumeration (enum ScopedCredentialType)
        3. 4.9.3 Unique Identifier for Credential (interface ScopedCredential)
        4. 4.9.4 Credential Descriptor (dictionary ScopedCredentialDescriptor)
        5. 4.9.5 Credential Transport enumeration (enum ExternalTransport)
        6. 4.9.6 Cryptographic Algorithm Identifier (type AlgorithmIdentifier)

1. Introduction

5. 5 WebAuthn Authenticator model
    1. 5.1 Authenticator operations
        1. 5.1.1 The authenticatorMakeCredential operation
        2. 5.1.2 The authenticatorGetAssertion operation
        3. 5.1.3 The authenticatorCancel operation
    2. 5.2 Signature Format
        1. 5.2.1 Authenticator data
        2. 5.2.2 Attestation data
        3. 5.2.3 Generating a signature
        4. 5.2.4 Verifying a signature
    3. 5.3 Credential Attestation
        1. 5.3.1 Attestation Statement Formats
        2. 5.3.2 Attestation Types
        3. 5.3.3 Generating an Attestation Object
        4. 5.3.4 Security Considerations
            1. 5.3.4.1 Privacy
            2. 5.3.4.2 Attestation Certificate and Attestation Certificate CA Compromise
            3. 5.3.4.3 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 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 Extending client processing
    5. 8.5 Extending authenticator processing
    6. 8.6 Example extension
9. 9 Pre-defined extensions
    1. 9.1 FIDO AppId
    2. 9.2 Transaction authorization
    3. 9.3 Authenticator Selection Extension
    4. 9.4 SupportedExtensions Extension
    5. 9.5 User Verification Index (UVI) Extension
    6. 9.6 Location Extension
    7. 9.7 User Verification Mode (UVM) Extension
10. 10 IANA Considerations
11. 11 Sample scenarios
    1. 11.1 Registration
    2. 11.2 Authentication
    3. 11.3 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, cryptographic scoped credentials by web applications, for the purpose of strongly authenticating users. A scoped credential is created and stored by an authenticator at the behest of a Relying Party, subject to user consent. Subsequently, the scoped credential can only be accessed by origins belonging to that Relying Party. This scoping is enforced jointly by conforming User Agents and authenticators. Additionally, privacy across Relying Parties is maintained; Relying Parties are not able to detect any properties, or even the existence, of credentials scoped to other Relying Parties.

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

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

1.1. Use Cases

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

1.1.1. Registration

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

1.1.2. Authentication

This section is not normative.

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

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

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

1.1. Use Cases

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

1.1.1. Registration

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

1.1.2. Authentication

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

1.1.3. Other use cases and configurations

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

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

2. Conformance

This specification defines criteria for a Conforming User Agent: A User Agent MUST behave as described in this specification in order to be considered conformant. Conforming User Agents MAY implement algorithms given in this specification in any way desired, so long as the end result is indistinguishable from the result that would be obtained by the specification's algorithms. A conforming User Agent MUST also be a conforming implementation of the IDL fragments of this specification, as described in the "Web IDL" specification. [WebIDL-1]

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

2.1. Dependencies

This specification relies on several other underlying specifications.

HTML

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

Web IDL

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

1.1.3. Other use cases and configurations

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

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

2. Conformance

This specification defines criteria for a Conforming User Agent: A User Agent MUST behave as described in this specification in order to be considered conformant. Conforming User Agents MAY implement algorithms given in this specification in any way desired, so long as the end result is indistinguishable from the result that would be obtained by the specification's algorithms. A conforming User Agent MUST also be a conforming implementation of the IDL fragments of this specification, as described in the "Web IDL" specification. [WebIDL-1]

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

2.1. Dependencies

This specification relies on several other underlying specifications.

HTML

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

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.

DOM

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

Web Cryptography API

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

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.

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

3. Terminology

ASCII case-insensitive match

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

Assertion

See Authentication Assertion.

Attestation

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

Attestation Certificate

A X.509 Certificate for the attestation key pair used by an Authenticator to attest to its manufacture and capabilities. At registration time, the authenticator uses the attestation private key to sign the Relying Party-specific credential public key (and additional data) that it generates and returns via the authenticatorMakeCredential operation. Relying Parties use the

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.

DOM

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

Web Cryptography API

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

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.

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

3. Terminology

ASCII case-insensitive match

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

Assertion

See Authentication Assertion.

Attestation

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

Attestation Certificate

A X.509 Certificate for the attestation key pair used by an Authenticator to attest to its manufacture and capabilities. At registration time, the authenticator uses the attestation private key to sign the Relying Party-specific credential public key (and additional data) that it generates and returns via the authenticatorMakeCredential operation. Relying Parties use the

attestation public key conveyed in the attestation certificate to verify the attestation signature. Note that in the case of self attestation, the authenticator has no distinct attestation key pair nor attestation certificate, see self attestation for details.

Authentication

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

Authentication Assertion

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

Authenticator

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

Authorization Gesture

Essentially the same as user verification.

Ceremony

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

Client

See Conforming User Agent.

Conforming User Agent

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

Credential Public Key

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

attestation public key conveyed in the attestation certificate to verify the attestation signature. Note that in the case of self attestation, the authenticator has no distinct attestation key pair nor attestation certificate, see self attestation for details.

Authentication

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

Authentication Assertion

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

Authenticator

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

Authorization Gesture

Essentially the same as user verification.

Ceremony

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

Client

See Conforming User Agent.

Conforming User Agent

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

Credential Public Key

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

attestation, the credential key pair is also used as the attestation key pair, see self attestation for details.

Registration

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

Relying Party

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

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

Relying Party Identifier

RP ID

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

Scoped Credential

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

User Consent

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

attestation, the credential key pair is also used as the attestation key pair, see self attestation for details.

Registration

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

Relying Party

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

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

Relying Party Identifier

RP ID

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

Scoped Credential

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

User Consent

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

User Verification

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

WebAuthn Client

See Conforming User Agent.

4. Web Authentication API

This section normatively specifies the API for creating and using scoped credentials. The basic idea is that the credentials belong to the user and are managed by an authenticator, with which the Relying Party interacts through the client (consisting of the browser and underlying OS platform). Scripts can (with the user's consent) request the browser to create a new credential for future use by the Relying Party. Scripts can also request the user's permission to perform authentication operations with an existing credential. All such operations are performed in the authenticator and are mediated by the browser and/or platform on the user's behalf. At no point does the script get access to the credentials themselves; it only gets information about the credentials in the form of objects.

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

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 Signature Format, the full origin of the requester is included, and signed over, in the attestation object produced when a new credential is created as well as in all assertions produced by WebAuthn credentials.

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

The client facilitates these security measures by providing correct origins and RP IDs to the authenticator for each operation. Since this

User Verification

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

WebAuthn Client

See Conforming User Agent.

4. Web Authentication API

This section normatively specifies the API for creating and using scoped credentials. The basic idea is that the credentials belong to the user and are managed by an authenticator, with which the Relying Party interacts through the client (consisting of the browser and underlying OS platform). Scripts can (with the user's consent) request the browser to create a new credential for future use by the Relying Party. Scripts can also request the user's permission to perform authentication operations with an existing credential. All such operations are performed in the authenticator and are mediated by the browser and/or platform on the user's behalf. At no point does the script get access to the credentials themselves; it only gets information about the credentials in the form of objects.

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

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 Signature Format, the full origin of the requester is included, and signed over, in the attestation object produced when a new credential is created as well as in all assertions produced by WebAuthn credentials.

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

The client facilitates these security measures by providing correct origins and RP IDs to the authenticator for each operation. Since this

is an integral part of the WebAuthn security model, user agents MUST only expose this API to callers in secure contexts, as defined in [secure-contexts].

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

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

4.1. WebAuthentication Interface

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

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

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

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

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

This method takes the following parameters:
* The accountInformation parameter specifies information about the user account for which the credential is being created. This is meant for later use by the authenticator when it needs to prompt the user to select a credential. An authenticator is only required to store one credential for any given value of accountInformation. Specifically, if an authenticator already has a credential for the specified value of id in accountInformation, and if this credential is not listed in the excludeList member of options, then after successful execution of this method:
    + Any calls to getAssertion() that do not specify allowList will not result in the older credential being offered to the user.
    + Any calls to getAssertion() that specify the older credential in the allowList may also not result in it being offered to the user.
* The cryptoParameters parameter supplies information about the desired properties of the credential to be created. The sequence is ordered from most preferred to least preferred. The platform makes a best effort to create the most preferred credential that it can.
* The attestationChallenge parameter contains a challenge intended to be used for generating the attestation object of the newly created

is an integral part of the WebAuthn security model, user agents MUST only expose this API to callers in secure contexts, as defined in [secure-contexts].

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

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

4.1. WebAuthentication Interface

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

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

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

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

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

This method takes the following parameters:
* The accountInformation parameter specifies information about the user account for which the credential is being created. This is meant for later use by the authenticator when it needs to prompt the user to select a credential. An authenticator is only required to store one credential for any given value of accountInformation. Specifically, if an authenticator already has a credential for the specified value of id in accountInformation, and if this credential is not listed in the excludeList member of options, then after successful execution of this method:
    + Any calls to getAssertion() that do not specify allowList will not result in the older credential being offered to the user.
    + Any calls to getAssertion() that specify the older credential in the allowList may also not result in it being offered to the user.
* The cryptoParameters parameter supplies information about the desired properties of the credential to be created. The sequence is ordered from most preferred to least preferred. The platform makes a best effort to create the most preferred credential that it can.
* The attestationChallenge parameter contains a challenge intended to be used for generating the attestation object of the newly created

credential.
* The optional options parameter specifies additional options, as described in 4.5 Additional options for Credential Generation (dictionary ScopedCredentialOptions).

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

1. If the timeoutSeconds 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 timeoutSeconds was not specified, then set adjustedTimeout to a platform-specific default.
2. Let promise be a new Promise. Return promise and start a timer for adjustedTimeout seconds. Then asynchronously continue executing the following steps. If any fatal error is encountered in this process other than the ones enumerated below, cancel the timer, reject promise with a DOMException whose name is "UnknownError", and terminate this algorithm.
3. Set callerOrigin to the current settings object's origin. If callerOrigin is an opaque origin, reject promise with a DOMException whose name is "NotAllowedError", and terminate this algorithm. Otherwise,
   + If the rpId member of options is not present, then set rpId to callerOrigin.
   + If the rpId member of options is present, then invoke the procedure used for relaxing the same-origin restriction by setting the document.domain attribute, using rpId as the given value but without changing the current document's domain. If no errors are thrown, set rpId to the value of host as computed by this procedure. Otherwise, reject promise with a DOMException whose name is "SecurityError", and terminate this algorithm.
4. Process each element of cryptoParameters using the following steps, to produce a new sequence normalizedParameters.
   + Let current be the currently selected element of cryptoParameters.
   + If current.type does not contain a ScopedCredentialType supported by this implementation, then stop processing current and move on to the next element in cryptoParameters.
   + Let normalizedAlgorithm be the result of normalizing an algorithm [WebCryptoAPI], with alg set to current.algorithm and op set to 'generateKey'. If an error occurs during this procedure, then stop processing current and move on to the next element in cryptoParameters.
   + Add a new object of type ScopedCredentialParameters to normalizedParameters, with type set to current.type and algorithm set to normalizedAlgorithm.
5. If normalizedAlgorithm is empty and cryptoParameters was not empty, cancel the timer started in step 2, reject promise with a DOMException whose name is "NotSupportedError", and terminate this algorithm.
6. If the extensions member of options is present, process any extensions supported by this client platform, to produce the extension data that needs to be sent to the authenticator. If an error is encountered while processing an extension, skip that extension and do not produce any extension data for it. Call the result of this processing clientExtensions.
7. Use attestationChallenge, callerOrigin and rpId, along with the token binding key associated with callerOrigin (if any), to create a ClientData structure representing this request. Choose a hash algorithm for hashAlg and compute the clientDataJSON and

credential.
* The optional options parameter specifies additional options, as described in 4.5 Additional options for Credential Generation (dictionary ScopedCredentialOptions).

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

1. If the timeoutSeconds 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 timeoutSeconds was not specified, then set adjustedTimeout to a platform-specific default.
2. Let promise be a new Promise. Return promise and start a timer for adjustedTimeout seconds. Then asynchronously continue executing the following steps. If any fatal error is encountered in this process other than the ones enumerated below, cancel the timer, reject promise with a DOMException whose name is "UnknownError", and terminate this algorithm.
3. Set callerOrigin to the current settings object's origin. If callerOrigin is an opaque origin, reject promise with a DOMException whose name is "NotAllowedError", and terminate this algorithm. Otherwise,
   + If the rpId member of options is not present, then set rpId to callerOrigin.
   + If the rpId member of options is present, then invoke the procedure used for relaxing the same-origin restriction by setting the document.domain attribute, using rpId as the given value but without changing the current document's domain. If no errors are thrown, set rpId to the value of host as computed by this procedure. Otherwise, reject promise with a DOMException whose name is "SecurityError", and terminate this algorithm.
4. Process each element of cryptoParameters using the following steps, to produce a new sequence normalizedParameters.
   + Let current be the currently selected element of cryptoParameters.
   + If current.type does not contain a ScopedCredentialType supported by this implementation, then stop processing current and move on to the next element in cryptoParameters.
   + Let normalizedAlgorithm be the result of normalizing an algorithm [WebCryptoAPI], with alg set to current.algorithm and op set to 'generateKey'. If an error occurs during this procedure, then stop processing current and move on to the next element in cryptoParameters.
   + Add a new object of type ScopedCredentialParameters to normalizedParameters, with type set to current.type and algorithm set to normalizedAlgorithm.
5. If normalizedAlgorithm is empty and cryptoParameters was not empty, cancel the timer started in step 2, reject promise with a DOMException whose name is "NotSupportedError", and terminate this algorithm.
6. If the extensions member of options is present, process any extensions supported by this client platform, to produce the extension data that needs to be sent to the authenticator. If an error is encountered while processing an extension, skip that extension and do not produce any extension data for it. Call the result of this processing clientExtensions.
7. Use attestationChallenge, callerOrigin and rpId, along with the token binding key associated with callerOrigin (if any), to create a ClientData structure representing this request. Choose a hash algorithm for hashAlg and compute the clientDataJSON and

clientDataHash.

8. Initialize issuedRequests and currentlyAvailableAuthenticators to empty lists.
9. For each authenticator currently available on this platform, add the authenticator to currentlyAvailableAuthenticators unless the attachment member of options is present. In that case, let attachment be attachment, and add the authenticator to currentlyAvailableAuthenticators if its attachment modality matches attachment.
10. For each authenticator in currentlyAvailableAuthenticators: asynchronously invoke the authenticatorMakeCredential operation on that authenticator with rpId, clientDataHash, accountInformation, normalizedParameters, excludeList and clientExtensions as parameters. Add a corresponding entry to issuedRequests.
    + For each credential C in the excludeList member of options that has a non-empty transports list, optionally use only the specified transports to test for the existence of C.
11. While issuedRequests is not empty, perform the following actions depending upon the adjustedTimeout timer and responses from the authenticators:
    + If the adjustedTimeout timer expires, then for each entry in issuedRequests invoke the authenticatorCancel operation on that authenticator and remove its entry from the list.
    + If any authenticator returns a status indicating that the user cancelled the operation, delete that authenticator's entry from issuedRequests. For each remaining entry in issuedRequests invoke the authenticatorCancel operation on that authenticator and remove its entry from the list.
    + If any authenticator returns an error status, delete the corresponding entry from issuedRequests.
    + If any authenticator indicates success:
        o Remove this authenticator's entry from issuedRequests.
        o Create a new ScopedCredentialInfo object named value and populate its fields with the values returned from the authenticator as well as the clientDataJSON computed earlier.
        o For each remaining entry in issuedRequests invoke the authenticatorCancel operation on that authenticator and remove its entry from the list.
        o Resolve promise with value and terminate this algorithm.
12. Reject promise with a DOMException whose name is "NotAllowedError", and terminate this algorithm.

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

4.1.2. Use an existing credential (getAssertion() method)

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

This method takes the following parameters:

* The assertionChallenge parameter contains a challenge that the selected authenticator is expected to sign to produce the assertion.
* The optional options parameter specifies additional options, as

clientDataHash.

8. Initialize issuedRequests and currentlyAvailableAuthenticators to empty lists.
9. For each authenticator currently available on this platform, add the authenticator to currentlyAvailableAuthenticators unless the attachment member of options is present. In that case, let attachment be attachment, and add the authenticator to currentlyAvailableAuthenticators if its attachment modality matches attachment.
10. For each authenticator in currentlyAvailableAuthenticators: asynchronously invoke the authenticatorMakeCredential operation on that authenticator with rpId, clientDataHash, accountInformation, normalizedParameters, excludeList and clientExtensions as parameters. Add a corresponding entry to issuedRequests.
    + For each credential C in the excludeList member of options that has a non-empty transports list, optionally use only the specified transports to test for the existence of C.
11. While issuedRequests is not empty, perform the following actions depending upon the adjustedTimeout timer and responses from the authenticators:
    + If the adjustedTimeout timer expires, then for each entry in issuedRequests invoke the authenticatorCancel operation on that authenticator and remove its entry from the list.
    + If any authenticator returns a status indicating that the user cancelled the operation, delete that authenticator's entry from issuedRequests. For each remaining entry in issuedRequests invoke the authenticatorCancel operation on that authenticator and remove its entry from the list.
    + If any authenticator returns an error status, delete the corresponding entry from issuedRequests.
    + If any authenticator indicates success:
        o Remove this authenticator's entry from issuedRequests.
        o Create a new ScopedCredentialInfo object named value and populate its fields with the values returned from the authenticator as well as the clientDataJSON computed earlier.
        o For each remaining entry in issuedRequests invoke the authenticatorCancel operation on that authenticator and remove its entry from the list.
        o Resolve promise with value and terminate this algorithm.
12. Reject promise with a DOMException whose name is "NotAllowedError", and terminate this algorithm.

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

4.1.2. Use an existing credential (getAssertion() method)

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

This method takes the following parameters:

* The assertionChallenge parameter contains a challenge that the selected authenticator is expected to sign to produce the assertion.
* The optional options parameter specifies additional options, as

described in 4.7 Additional options for Assertion Generation (dictionary AssertionOptions).

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

1. If the timeoutSeconds 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 timeoutSeconds is not present, then set adjustedTimeout to a platform-specific default.
2. Let promise be a new Promise. Return promise and start a timer for adjustedTimeout seconds. Then asynchronously continue executing the following steps. If any fatal error is encountered in this process other than the ones enumerated below, cancel the timer, reject promise with a DOMException whose name is "UnknownError", and terminate this algorithm.
3. Set callerOrigin to the current settings object's origin. If callerOrigin is an opaque origin, reject promise with a DOMException whose name is "NotAllowedError", and terminate this algorithm. Otherwise,
   + If the rpId member of options is not present, then set rpId to callerOrigin.
   + If the rpId member of options is present, then invoke the procedure used for relaxing the same-origin restriction by setting the document.domain attribute, using rpId as the given value but without changing the current document's domain. If no errors are thrown, set rpId to the value of host as computed by this procedure. Otherwise, reject promise with a DOMException whose name is "SecurityError", and terminate this algorithm.
4. If the extensions member of options is present, process any extensions supported by this client platform, to produce the extension data that needs to be sent to the authenticator. If an error is encountered while processing an extension, skip that extension and do not produce any extension data for it. Call the result of this processing clientExtensions.
5. Use assertionChallenge, callerOrigin and rpId, along with the token binding key associated with callerOrigin (if any), to create a ClientData structure representing this request. Choose a hash algorithm for hashAlg and compute the clientDataJSON and clientDataHash.
6. Initialize issuedRequests to an empty list.
7. For each authenticator currently available on this platform, perform the following steps:
   + If the allowList member of options is empty, let credentialList be an empty list. Otherwise, execute a platform-specific procedure to determine which, if any, credentials listed in allowList might be present on this authenticator, and set credentialList to this filtered list. If no such filtering is possible, set credentialList to an empty list.
   + For each credential C within the credentialList that has a non-empty transports list, optionally use only the specified transports to get assertions using credential C.
   + If the above filtering process concludes that none of the credentials on the allowList can possibly be on this authenticator, do not perform any of the following steps for this authenticator, and proceed to the next authenticator (if any).
   + Asynchronously invoke the authenticatorGetAssertion operation on this authenticator with rpId, clientDataHash,

described in 4.7 Additional options for Assertion Generation (dictionary AssertionOptions).

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

1. If the timeoutSeconds 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 timeoutSeconds is not present, then set adjustedTimeout to a platform-specific default.
2. Let promise be a new Promise. Return promise and start a timer for adjustedTimeout seconds. Then asynchronously continue executing the following steps. If any fatal error is encountered in this process other than the ones enumerated below, cancel the timer, reject promise with a DOMException whose name is "UnknownError", and terminate this algorithm.
3. Set callerOrigin to the current settings object's origin. If callerOrigin is an opaque origin, reject promise with a DOMException whose name is "NotAllowedError", and terminate this algorithm. Otherwise,
   + If the rpId member of options is not present, then set rpId to callerOrigin.
   + If the rpId member of options is present, then invoke the procedure used for relaxing the same-origin restriction by setting the document.domain attribute, using rpId as the given value but without changing the current document's domain. If no errors are thrown, set rpId to the value of host as computed by this procedure. Otherwise, reject promise with a DOMException whose name is "SecurityError", and terminate this algorithm.
4. If the extensions member of options is present, process any extensions supported by this client platform, to produce the extension data that needs to be sent to the authenticator. If an error is encountered while processing an extension, skip that extension and do not produce any extension data for it. Call the result of this processing clientExtensions.
5. Use assertionChallenge, callerOrigin and rpId, along with the token binding key associated with callerOrigin (if any), to create a ClientData structure representing this request. Choose a hash algorithm for hashAlg and compute the clientDataJSON and clientDataHash.
6. Initialize issuedRequests to an empty list.
7. For each authenticator currently available on this platform, perform the following steps:
   + If the allowList member of options is empty, let credentialList be an empty list. Otherwise, execute a platform-specific procedure to determine which, if any, credentials listed in allowList might be present on this authenticator, and set credentialList to this filtered list. If no such filtering is possible, set credentialList to an empty list.
   + For each credential C within the credentialList that has a non-empty transports list, optionally use only the specified transports to get assertions using credential C.
   + If the above filtering process concludes that none of the credentials on the allowList can possibly be on this authenticator, do not perform any of the following steps for this authenticator, and proceed to the next authenticator (if any).
   + Asynchronously invoke the authenticatorGetAssertion operation on this authenticator with rpId, clientDataHash,

credentialList, and clientExtensions as parameters.
  + Add an entry to issuedRequests, corresponding to this request.

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

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

4.2. Information about Scoped Credential (interface ScopedCredentialInfo)

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

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

The clientData member contains the clientDataJSON (see 5.2 Signature Format) passed to the authenticator by the client in order to generate this credential. The exact JSON encoding must be preserved as a cryptographic hash (clientDataHash) has been computed over it.

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

credentialList, and clientExtensions as parameters.
  + Add an entry to issuedRequests, corresponding to this request.

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

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

4.2. Information about Scoped Credential (interface ScopedCredentialInfo)

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

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

The clientData member contains the clientDataJSON (see 5.2 Signature Format) passed to the authenticator by the client in order to generate this credential. The exact JSON encoding must be preserved as a cryptographic hash (clientDataHash) has been computed over it.

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

4.3. User Account Information (dictionary Account)

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

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

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

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

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

The name member contains a detailed name for the account, such as "john.p.smith@example.com".

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

4.4. Parameters for Credential Generation (dictionary ScopedCredentialParameters)

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

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

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

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

4.5. Additional options for Credential Generation (dictionary ScopedCredentialOptions)

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


4.3. User Account Information (dictionary Account)

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

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

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

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

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

The name member contains a detailed name for the account, such as "john.p.smith@example.com".

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

4.4. Parameters for Credential Generation (dictionary ScopedCredentialParameters)

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

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

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

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

4.5. Additional options for Credential Generation (dictionary ScopedCredentialOptions)

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

};

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

* The timeoutSeconds parameter specifies a time, in seconds, that the caller is willing to wait for the call to complete. This is treated as a hint, and may be overridden by the platform.
* The rpId parameter explicitly specifies the RP ID that the credential should be associated with. If it is omitted, the RP ID will be set to the current settings object's origin.
* The excludeList parameter is intended for use by Relying Parties that wish to limit the creation of multiple credentials for the same account on a single authenticator. The platform is requested to return an error if the new credential would be created on an authenticator that also contains one of the credentials enumerated in this parameter.
* The extensions parameter contains additional parameters requesting additional processing by the client and authenticator. For example, the caller may request that only authenticators with certain capabilities be used to create the credential, or that additional information be returned in the attestation object. Alternatively, the caller may specify an additional message that they would like the authenticator to display to the user. Extensions are defined in 8 WebAuthn Extensions.
* The attachment parameter contains authenticator attachment descriptions, which are used as an additional constraint on which authenticators are eligible to participate in a 4.1.1 Create a new credential (makeCredential() method) or 4.1.2 Use an existing credential (getAssertion() method) operation. See 4.5.1 Credential Attachment enumeration (enum Attachment) for a description of the attachment values and their meanings.

4.5.1. Credential Attachment enumeration (enum Attachment)

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

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

* 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

};

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

* The timeoutSeconds parameter specifies a time, in seconds, that the caller is willing to wait for the call to complete. This is treated as a hint, and may be overridden by the platform.
* The rpId parameter explicitly specifies the RP ID that the credential should be associated with. If it is omitted, the RP ID will be set to the current settings object's origin.
* The excludeList parameter is intended for use by Relying Parties that wish to limit the creation of multiple credentials for the same account on a single authenticator. The platform is requested to return an error if the new credential would be created on an authenticator that also contains one of the credentials enumerated in this parameter.
* The extensions parameter contains additional parameters requesting additional processing by the client and authenticator. For example, the caller may request that only authenticators with certain capabilities be used to create the credential, or that additional information be returned in the attestation object. Alternatively, the caller may specify an additional message that they would like the authenticator to display to the user. Extensions are defined in 8 WebAuthn Extensions.
* The attachment parameter contains authenticator attachment descriptions, which are used as an additional constraint on which authenticators are eligible to participate in a 4.1.1 Create a new credential (makeCredential() method) or 4.1.2 Use an existing credential (getAssertion() method) operation. See 4.5.1 Credential Attachment enumeration (enum Attachment) for a description of the attachment values and their meanings.

4.5.1. Credential Attachment enumeration (enum Attachment)

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

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

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

This distinction is important because there are use-cases where only platform authenticators are acceptable to a Relying Party, and conversely ones where only roaming authenticators are employed. As a concrete example of the former, a credential on a platform

authenticator may be used by Relying Parties to quickly and conveniently reauthenticate the user with a minimum of friction, e.g., the user will not have to dig around in their pocket for their key fob or phone. As a concrete example of the latter, when the user is accessing the Relying Party from a given client for the first time, they may be required to use a roaming authenticator which was originally registered with the Relying Party using a different client.

4.6. Web Authentication Assertion (interface AuthenticationAssertion)

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

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

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

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

The authenticatorData member contains the serialized data returned by the authenticator. See 5.2.1 Authenticator data.

The signature member contains the raw signature returned from the authenticator. See 5.2.3 Generating a signature.

4.7. Additional options for Assertion Generation (dictionary AssertionOptions)

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

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

* The optional timeoutSeconds parameter specifies a time, in seconds, that the caller is willing to wait for the call to complete. This is treated as a hint, and may be overridden by the platform.
* The optional rpId parameter specifies the rpId claimed by the caller. If it is omitted, it will be assumed to be equal to the current settings object's origin.
* The optional allowList member contains a list of credentials acceptable to the caller, in order of the caller's preference.
* The optional extensions parameter contains additional parameters requesting additional processing by the client and authenticator. For example, if transaction confirmation is sought from the user, then the prompt string would be included in an extension. Extensions are defined in a companion specification.

authenticator may be used by Relying Parties to quickly and conveniently reauthenticate the user with a minimum of friction, e.g., the user will not have to dig around in their pocket for their key fob or phone. As a concrete example of the latter, when the user is accessing the Relying Party from a given client for the first time, they may be required to use a roaming authenticator which was originally registered with the Relying Party using a different client.

4.6. Web Authentication Assertion (interface AuthenticationAssertion)

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

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

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

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

The authenticatorData member contains the serialized data returned by the authenticator. See 5.2.1 Authenticator data.

The signature member contains the raw signature returned from the authenticator. See 5.2.3 Generating a signature.

4.7. Additional options for Assertion Generation (dictionary AssertionOptions)

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

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

* The optional timeoutSeconds parameter specifies a time, in seconds, that the caller is willing to wait for the call to complete. This is treated as a hint, and may be overridden by the platform.
* The optional rpId parameter specifies the rpId claimed by the caller. If it is omitted, it will be assumed to be equal to the current settings object's origin.
* The optional allowList member contains a list of credentials acceptable to the caller, in order of the caller's preference.
* The optional extensions parameter contains additional parameters requesting additional processing by the client and authenticator. For example, if transaction confirmation is sought from the user, then the prompt string would be included in an extension. Extensions are defined in a companion specification.

4.8. Authentication Assertion Extensions (dictionary AuthenticationExtensions)

```
dictionary AuthenticationExtensions {
};
```

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

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

4.9. Supporting Data Structures

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

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

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

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

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

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

The hashAlg member specifies the hash algorithm used to compute clientDataHash. Use "S256" for SHA-256, "S384" for SHA384, "S512" for SHA512, and "SM3" for SM3 (see 10 IANA Considerations). This algorithm is chosen by the client at its sole discretion.

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

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

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

clientDataJSON

This is the UTF-8 encoded JSON serialization [RFC7159] of a ClientData dictionary. Any valid JSON serialization may be used

4.8. Authentication Assertion Extensions (dictionary AuthenticationExtensions)

dictionary AuthenticationExtensions {
};

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

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

4.9. Supporting Data Structures

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

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

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

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

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

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

The hashAlg member specifies the hash algorithm used to compute clientDataHash. Use "S256" for SHA-256, "S384" for SHA384, "S512" for SHA512, and "SM3" for SM3 (see 10 IANA Considerations). This algorithm is chosen by the client at its sole discretion.

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

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

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

clientDataJSON

This is the UTF-8 encoded JSON serialization [RFC7159] of a ClientData dictionary. Any valid JSON serialization may be used

by the client. This specification imposes no canonicalization requirements.

clientDataHash

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

4.9.2. Credential Type enumeration (enum ScopedCredentialType)

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

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

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

4.9.3. Unique Identifier for Credential (interface ScopedCredential)

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

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

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

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

4.9.4. Credential Descriptor (dictionary ScopedCredentialDescriptor)

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

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

by the client. This specification imposes no canonicalization requirements.

clientDataHash

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

4.9.2. Credential Type enumeration (enum ScopedCredentialType)

enum ScopedCredentialType {
    "ScopedCred"
};

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

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

4.9.3. Unique Identifier for Credential (interface ScopedCredential)

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

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

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

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

4.9.4. Credential Descriptor (dictionary ScopedCredentialDescriptor)

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

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

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

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

4.9.5. Credential Transport enumeration (enum ExternalTransport)

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

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

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

4.9.6. Cryptographic Algorithm Identifier (type AlgorithmIdentifier)

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

5. WebAuthn Authenticator model

The API defined in this specification implies a specific abstract functional model for an authenticator. This section describes the authenticator model. Client platforms may implement and expose this abstract model in any way desired. For instance, 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. The overall requirement is that 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.

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

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

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

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

4.9.5. Credential Transport enumeration (enum ExternalTransport)

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

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

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

4.9.6. Cryptographic Algorithm Identifier (type AlgorithmIdentifier)

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

5. WebAuthn Authenticator model

The API defined in this specification implies a specific abstract functional model for an authenticator. This section describes the authenticator model. Client platforms may implement and expose this abstract model in any way desired. For instance, 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. The overall requirement is that 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.

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

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

certain properties of the authenticator, such as certification level and strength of key protection, using information from other sources.

5.1. Authenticator operations

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

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

5.1.1. The authenticatorMakeCredential operation

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

When this operation is invoked, the authenticator must perform the following procedure:
* Check if all the supplied parameters are syntactically well-formed and of the correct length. If not, return an error code equivalent to UnknownError and terminate the operation.
* Check if at least one of the specified combinations of ScopedCredentialType and cryptographic parameters is supported. If not, return an error code equivalent to NotSupportedError and terminate the operation.
* Check if a credential matching any of the supplied ScopedCredential identifiers is present on this authenticator. If so, return an error code equivalent to NotAllowedError and terminate the operation.
* Prompt the user for consent to create a new credential. The prompt for obtaining this consent is shown by the authenticator if it has its own output capability, or by the user agent otherwise. If the user denies consent, return an error code equivalent to NotAllowedError and terminate the operation.
* Once user consent has been obtained, generate a new credential object:
  + Generate a set of cryptographic keys using the most preferred combination of ScopedCredentialType and cryptographic parameters supported by this authenticator.
  + Generate an identifier for this credential, such that this identifier is globally unique with high probability across all credentials with the same type across all authenticators.
  + Associate the credential with the specified RP ID and the user's account identifier id.
  + Delete any older credentials with the same RP ID and id that are stored locally in the authenticator.

certain properties of the authenticator, such as certification level and strength of key protection, using information from other sources.

5.1. Authenticator operations

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

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

5.1.1. The authenticatorMakeCredential operation

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

When this operation is invoked, the authenticator must perform the following procedure:
* Check if all the supplied parameters are syntactically well-formed and of the correct length. If not, return an error code equivalent to UnknownError and terminate the operation.
* Check if at least one of the specified combinations of ScopedCredentialType and cryptographic parameters is supported. If not, return an error code equivalent to NotSupportedError and terminate the operation.
* Check if a credential matching any of the supplied ScopedCredential identifiers is present on this authenticator. If so, return an error code equivalent to NotAllowedError and terminate the operation.
* Prompt the user for consent to create a new credential. The prompt for obtaining this consent is shown by the authenticator if it has its own output capability, or by the user agent otherwise. If the user denies consent, return an error code equivalent to NotAllowedError and terminate the operation.
* Once user consent has been obtained, generate a new credential object:
  + Generate a set of cryptographic keys using the most preferred combination of ScopedCredentialType and cryptographic parameters supported by this authenticator.
  + Generate an identifier for this credential, such that this identifier is globally unique with high probability across all credentials with the same type across all authenticators.
  + Associate the credential with the specified RP ID and the user's account identifier id.
  + Delete any older credentials with the same RP ID and id that are stored locally in the authenticator.

* If any error occurred while creating the new credential object, return an error code equivalent to UnknownError and terminate the operation.
* Process all the supported extensions requested by the client, and generate an authenticatorData structure with attestation data as specified in 5.2.1 Authenticator data. Use this authenticatorData and the clientDataHash received from the client to create an attestation object for the new credential using the procedure specified in 5.3.3 Generating an Attestation Object. For more details on attestation, see 5.3 Credential Attestation.

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

5.1.2. The authenticatorGetAssertion operation

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

When this method is invoked, the authenticator must perform the following procedure:
* Check if all the supplied parameters are syntactically well-formed and of the correct length. If not, return an error code equivalent to UnknownError and terminate the operation.
* If a list of credentials was supplied by the client, filter it by removing those credentials that are not present on this authenticator. If no list was supplied, create a list with all credentials stored for the caller's RP ID (as determined by an exact match of the RP ID).
* If the previous step resulted in an empty list, return an error code equivalent to NotAllowedError and terminate the operation.
* Prompt the user to select a credential from among the above list. Obtain user consent for using this credential. The prompt for obtaining this consent may be shown by the authenticator if it has its own output capability, or by the user agent otherwise.
* Process all the supported extensions requested by the client, and generate an authenticatorData structure without attestation data as specified in 5.2.1 Authenticator data. Use this authenticatorData and the clientDataHash received from the client to generate an assertion signature using the private key of the selected credential, as specified in 5.2.3 Generating a signature.
* If any error occurred while generating the assertion signature, return an error code equivalent to UnknownError and terminate the operation.

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

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

* If any error occurred while creating the new credential object, return an error code equivalent to UnknownError and terminate the operation.
* Process all the supported extensions requested by the client, and generate an authenticatorData structure with attestation data as specified in 5.2.1 Authenticator data. Use this authenticatorData and the clientDataHash received from the client to create an attestation object for the new credential using the procedure specified in 5.3.3 Generating an Attestation Object. For more details on attestation, see 5.3 Credential Attestation.

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

5.1.2. The authenticatorGetAssertion operation

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

When this method is invoked, the authenticator must perform the following procedure:
* Check if all the supplied parameters are syntactically well-formed and of the correct length. If not, return an error code equivalent to UnknownError and terminate the operation.
* If a list of credentials was supplied by the client, filter it by removing those credentials that are not present on this authenticator. If no list was supplied, create a list with all credentials stored for the caller's RP ID (as determined by an exact match of the RP ID).
* If the previous step resulted in an empty list, return an error code equivalent to NotAllowedError and terminate the operation.
* Prompt the user to select a credential from among the above list. Obtain user consent for using this credential. The prompt for obtaining this consent may be shown by the authenticator if it has its own output capability, or by the user agent otherwise.
* Process all the supported extensions requested by the client, and generate an authenticatorData structure without attestation data as specified in 5.2.1 Authenticator data. Use this authenticatorData and the clientDataHash received from the client to generate an assertion signature using the private key of the selected credential, as specified in 5.2.3 Generating a signature.
* If any error occurred while generating the assertion signature, return an error code equivalent to UnknownError and terminate the operation.

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

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

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

5.1.3. 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.

5.2. Signature Format

WebAuthn signatures 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.

The components of a system using WebAuthn can be divided into three layers:

1. The Relying Party (RP), which uses the WebAuthn services. The RP consists of a server component and a web-application running in a browser.
2. The WebAuthn Client platform, which consists of the User Agent and the OS and device on which it executes.
3. The Authenticator itself, which provides key management and cryptographic signatures. This may be embedded in the WebAuthn client, or housed in a separate device entirely. In the latter case, the interface between the WebAuthn client and the authenticator is a separately-defined protocol. 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.

This specification defines the common signature format shared by all the above layers. This includes how the different contextual bindings are encoded, signed over, and delivered to the RP.

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.

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

5.1.3. 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.

5.2. Signature Format

WebAuthn signatures 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.

The components of a system using WebAuthn can be divided into three layers:

1. The Relying Party (RP), which uses the WebAuthn services. The RP consists of a server component and a web-application running in a browser.
2. The WebAuthn Client platform, which consists of the User Agent and the OS and device on which it executes.
3. The Authenticator itself, which provides key management and cryptographic signatures. This may be embedded in the WebAuthn client, or housed in a separate device entirely. In the latter case, the interface between the WebAuthn client and the authenticator is a separately-defined protocol. 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.

This specification defines the common signature format shared by all the above layers. This includes how the different contextual bindings are encoded, signed over, and delivered to the RP.

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.

The 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 client data must be signed over, but an authenticator is otherwise not interested in its contents. To save bandwidth and processing requirements on the authenticator, the client hashes the ClientData and sends only the result to the authenticator. The authenticator signs over the combination of this clientDataHash, and its own authenticator data.

5.2.1. Authenticator data

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

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

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

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

4 Signature counter (signCount), 32-bit unsigned big-endian integer.
variable (if present) Attestation data (if present). See 5.3.3 Generating an Attestation Object for details. Its length depends on the length of the credential public key and credential ID being attested.
variable (if present) Extension-defined authenticator data. This is a CBOR [RFC7049] map with extension identifiers as keys, and extension authenticator data values as values. See 8 WebAuthn Extensions for details.

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

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

The 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 client data must be signed over, but an authenticator is otherwise not interested in its contents. To save bandwidth and processing requirements on the authenticator, the client hashes the ClientData and sends only the result to the authenticator. The authenticator signs over the combination of this clientDataHash, and its own authenticator data.

5.2.1. Authenticator data

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

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

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

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

4 Signature counter (signCount), 32-bit unsigned big-endian integer.
variable (if present) Attestation data (if present). See 5.3.3 Generating an Attestation Object for details. Its length depends on the length of the credential public key and credential ID being attested.
variable (if present) Extension-defined authenticator data. This is a CBOR [RFC7049] map with extension identifiers as keys, and extension authenticator data values as values. See 8 WebAuthn Extensions for details.

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

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

flags byte SHALL be set to zero.

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

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

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

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

5.2.2. Attestation data

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

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

alg

This is the name of the signature algorithm associated with the credential private key, expressed as defined in [RFC7518] section 3.1. Specifically, the following values are supported: "ES256", "ES384", "ES512", "RS256", "RS384", "RS512", "PS256", "PS384" and "PS512".

(public key fields)

These fields contain the public key, expressed in the format defined by [RFC7518] section 6. Specifically, for ECC keys, the x and y fields are present as defined in [RFC7518] sections 6.2.1.2 and 6.2.1.3, and for RSA keys, the n and e fields are present as defined in [RFC7518] sections 6.3.1.1 and 6.3.1.2.

5.2.3. Generating a signature

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

flags byte SHALL be set to zero.

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

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

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

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

5.2.2. Attestation data

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

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

alg

This is the name of the signature algorithm associated with the credential private key, expressed as defined in [RFC7518] section 3.1. Specifically, the following values are supported: "ES256", "ES384", "ES512", "RS256", "RS384", "RS512", "PS256", "PS384" and "PS512".

(public key fields)

These fields contain the public key, expressed in the format defined by [RFC7518] section 6. Specifically, for ECC keys, the x and y fields are present as defined in [RFC7518] sections 6.2.1.2 and 6.2.1.3, and for RSA keys, the n and e fields are present as defined in [RFC7518] sections 6.3.1.1 and 6.3.1.2.

5.2.3. Generating a signature

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

user has consented to a specific transaction. Thus, an assertion signature asserts the identity of the requester, and provides additional information that might be useful to the caller, such as the means by which user consent was provided, and the prompt that was shown to the user by the authenticator.

Both types of signature must assert the integrity of both the client data and the authenticator data. Thus, in both cases the authenticator computes a signature over a combination of the clientDataHash and the authenticatorData.

To generate either type of signature, the authenticator must first select an appropriate private key for the signature. Then, the authenticator concatenates the authenticatorData and clientDataHash, and signs the result using the selected private key as shown in the figure below.
[fido-signature-formats-figure2.svg] Generating a signature on the authenticator.

A simple, undelimited concatenation is safe to use here because the authenticatorData describes its own length. The clientDataHash (which potentially has a variable length) is always the last element.

5.2.4. Verifying a signature

This section specifies the algorithm for verifying any signature generated using the method in 5.2.3 Generating a signature.

To verify a signature S given a claimed clientDataJSON C and a claimed authenticatorData data using a given public key, the Relying Party shall:
1. Perform JSON decoding on C to extract the ClientData used for the signature.
2. Verify that the challenge in the ClientData matches the challenge that was sent to the authenticator.
3. Verify that the origin in the ClientData matches the Relying Party's origin.
4. Verify that the tokenBinding (if present) in the ClientData matches the token binding ID for the TLS connection over which the signature was obtained.
5. Verify that the extensions in the ClientData is a proper subset of the extensions requested by the RP.
6. Verify that the RP ID hash in data is indeed the SHA-256 hash of the RP ID expected by the RP.
7. Compute the clientDataHash over C using the hashAlg algorithm found in the ClientData structure.
8. Use the given public key to verify that S is a valid signature over the binary concatenation of data and the clientDataHash computed above.

If all the above steps succeed, then the signature is valid, otherwise it is invalid.

5.3. Credential Attestation

Authenticators must also provide some form of attestation. The basic requirement is that the authenticator can produce, for each credential public key, attestation information that can be verified by a Relying Party. Typically, this information contains a signature by an attestation private key over the attested credential public key and a challenge, as well as a certificate or similar information providing provenance information for the attestation public key, enabling a trust

user has consented to a specific transaction. Thus, an assertion signature asserts the identity of the requester, and provides additional information that might be useful to the caller, such as the means by which user consent was provided, and the prompt that was shown to the user by the authenticator.

Both types of signature must assert the integrity of both the client data and the authenticator data. Thus, in both cases the authenticator computes a signature over a combination of the clientDataHash and the authenticatorData.

To generate either type of signature, the authenticator must first select an appropriate private key for the signature. Then, the authenticator concatenates the authenticatorData and clientDataHash, and signs the result using the selected private key as shown in the figure below.
[fido-signature-formats-figure2.svg] Generating a signature on the authenticator.

A simple, undelimited concatenation is safe to use here because the authenticatorData describes its own length. The clientDataHash (which potentially has a variable length) is always the last element.

5.2.4. Verifying a signature

This section specifies the algorithm for verifying any signature generated using the method in 5.2.3 Generating a signature.

To verify a signature S given a claimed clientDataJSON C and a claimed authenticatorData data using a given public key, the Relying Party shall:
1. Perform JSON decoding on C to extract the ClientData used for the signature.
2. Verify that the challenge in the ClientData matches the challenge that was sent to the authenticator.
3. Verify that the origin in the ClientData matches the Relying Party's origin.
4. Verify that the tokenBinding (if present) in the ClientData matches the token binding ID for the TLS connection over which the signature was obtained.
5. Verify that the extensions in the ClientData is a proper subset of the extensions requested by the RP.
6. Verify that the RP ID hash in data is indeed the SHA-256 hash of the RP ID expected by the RP.
7. Compute the clientDataHash over C using the hashAlg algorithm found in the ClientData structure.
8. Use the given public key to verify that S is a valid signature over the binary concatenation of data and the clientDataHash computed above.

If all the above steps succeed, then the signature is valid, otherwise it is invalid.

5.3. Credential Attestation

Authenticators must also provide some form of attestation. The basic requirement is that the authenticator can produce, for each credential public key, attestation information that can be verified by a Relying Party. Typically, this information contains a signature by an attestation private key over the attested credential public key and a challenge, as well as a certificate or similar information providing provenance information for the attestation public key, enabling a trust

decision to be made. However, if an attestation key pair is not available, then the authenticator MUST perform self attestation of the credential public key with the corresponding credential private key. All this information is returned by the authenticator any time a new credential is generated, in the form of an attestation object.

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

1. The attestation statement format is the manner in which the signature is represented and the various contextual bindings are incorporated into the attestation statement by the authenticator. In other words, this defines the syntax of the statement. Various existing devices and platforms (such as TPMs and the Android OS) have previously defined attestation statement formats. This specification supports a variety of such formats in an extensible way, as defined in 5.3.1 Attestation Statement Formats.
2. The attestation type defines the semantics of the attestation statement and its underlying trust model. It defines how a Relying Party establishes trust in a particular attestation statement, after verifying that it is cryptographically valid.

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

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

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

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

5.3.1. Attestation Statement Formats

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

* Its attestation format identifier.
* The set of attestation types supported by the format.
* The syntax of an attestation statement produced in this format.

decision to be made. However, if an attestation key pair is not available, then the authenticator MUST perform self attestation of the credential public key with the corresponding credential private key. All this information is returned by the authenticator any time a new credential is generated, in the form of an attestation object.

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

1. The attestation statement format is the manner in which the signature is represented and the various contextual bindings are incorporated into the attestation statement by the authenticator. In other words, this defines the syntax of the statement. Various existing devices and platforms (such as TPMs and the Android OS) have previously defined attestation statement formats. This specification supports a variety of such formats in an extensible way, as defined in 5.3.1 Attestation Statement Formats.
2. The attestation type defines the semantics of the attestation statement and its underlying trust model. It defines how a Relying Party establishes trust in a particular attestation statement, after verifying that it is cryptographically valid.

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

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

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

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

5.3.1. Attestation Statement Formats

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

* Its attestation format identifier.
* The set of attestation types supported by the format.
* The syntax of an attestation statement produced in this format.

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

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

5.3.2. 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 5.3.4.1 Privacy for futher information.

Self Attestation

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

Privacy CA

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

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

Direct Anonymous Attestation (DAA)

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

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

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

5.3.2. 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 5.3.4.1 Privacy for futher information.

Self Attestation

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

Privacy CA

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

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

Direct Anonymous Attestation (DAA)

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

5.3.3. Generating an Attestation Object

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

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

The authenticator MUST then concatenate this authenticatorData and the client-supplied clientDataHash as specified in 5.2.3 Generating a signature to form attToBeSigned. It must then run the signing procedure for the desired attestation statement format with attToBeSigned 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 comprising the following fields:

format

The attestation format identifier associated with the attestation statement.

authenticatorData

The authenticator data used to generate the attestation statement.

attestation

The attestation statement constructed above.

5.3.4. Security Considerations

5.3.4.1. Privacy

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

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

5.3.4.2. Attestation Certificate and Attestation Certificate CA Compromise

When an intermediate CA or a root CA used for issuing attestation

5.3.3. Generating an Attestation Object

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

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

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

Finally, the authenticator MUST construct the attestation object as a CBOR map comprising the following fields:

fmt

The attestation format identifier associated with the attestation statement.

authData

The authenticator data used to generate the attestation statement.

attStmt

The attestation statement constructed above.

5.3.4. Security Considerations

5.3.4.1. Privacy

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

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

5.3.4.2. Attestation Certificate and Attestation Certificate CA Compromise

When an intermediate CA or a root CA used for issuing attestation

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

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

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

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

5.3.4.3. Attestation Certificate Hierarchy

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

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

6. Relying Party Operations

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

6.1. Registering a new credential

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

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

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

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

5.3.4.3. Attestation Certificate Hierarchy

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

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

6. Relying Party Operations

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

6.1. Registering a new credential

When requested to register a new credential with a ScopedCredentialInfo structure, a Relying Party must proceed as follows:

1. Perform JSON decoding on the clientData field of the ScopedCredentialInfo object to extract the ClientData used for the credential's attestation.
2. Verify that the challenge in the ClientData matches the challenge that was sent to the authenticator in the makeCredential() call.
3. Verify that the origin in the ClientData matches the Relying Party's origin.
4. Verify that the tokenBinding in the ClientData matches the token binding ID for the TLS connection over which the attestation was obtained.
5. Verify that the extensions in the ClientData is a proper subset of the extensions requested by the RP.
6. Compute the clientDataHash over clientData using the hashAlg algorithm found in the ClientData structure.
7. Perform CBOR decoding on the attestationObject field of the ScopedCredentialInfo structure to obtain the attestation statement format fmt, the authenticator data data, and the attestation statement stmt.
8. Verify that the RP ID hash in data is indeed the SHA-256 hash of the RP ID expected by the RP.
9. Perform an ASCII case-insensitive match on fmt to determine the attestation statement format.
10. Using the verification process for the above attestation statement format, validate that stmt is a valid attestation statement for authenticator data data and the clientDataHash computed in step 6.
11. If validation is successful, obtain a list of acceptable trust anchors (attestation root certificates or DAA root keys) for that attestation type and attestation statement format fmt, from a trusted source or from policy. For example, the FIDO Metadata Service [FIDOMetadataService] provides one way to access such information, using the AAGUID in data.
12. Verify the trustworthiness of the attestation using the outputs of the verification process in step 10 as follows:
    + If self-attestation was used, check if self-attestation is acceptable under Relying Party policy.
    + If DAA was used, verify that the DAA key used is in the set of acceptable trust anchors obtained above.
    + Otherwise, use the X.509 certificates returned by the verification process to verify that the attestation public key correctly chains up to an acceptable root certificate.
13. If the attestation statement was correctly verified and found to be trustworthy, then register the new credential by associating the credential ID and credential public key found in data with the Relying Party user on whose behalf the makeCredential() operation was requested.
14. If the attestation statement was correctly verified but could not be established to be trustworthy, the Relying Party SHOULD reject the registration operation. 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 5.3.2 Attestation Types). If doing so, the Relying Party is asserting there is no cryptographic proof that the Scoped Credential has been generated by a particular Authenticator model. See [FIDOSecRef] and [UAFProtocol] for a more detailed discussion.
15. If the attestation statement could not be correctly verified, the Relying Party MUST reject the registration operation.

Verification of attestation objects requires that the Relying Party has

When requested to register a new credential with a ScopedCredentialInfo structure, a Relying Party must proceed as follows:

1. Perform JSON decoding on the clientData field of the ScopedCredentialInfo object to extract the ClientData used for the credential's attestation.
2. Verify that the challenge in the ClientData matches the challenge that was sent to the authenticator in the makeCredential() call.
3. Verify that the origin in the ClientData matches the Relying Party's origin.
4. Verify that the tokenBinding in the ClientData matches the token binding ID for the TLS connection over which the attestation was obtained.
5. Verify that the extensions in the ClientData is a proper subset of the extensions requested by the RP.
6. Compute the clientDataHash over clientData using the hashAlg algorithm found in the ClientData structure.
7. Perform CBOR decoding on the attestationObject field of the ScopedCredentialInfo structure to obtain the attestation statement format fmt, the authenticator data authData, and the attestation statement attStmt.
8. Verify that the RP ID hash in authData is indeed the SHA-256 hash of the RP ID expected by the RP.
9. Perform an ASCII case-insensitive match on fmt to determine the attestation statement format.
10. Using the verification process for the above attestation statement format, validate that attStmt is a valid attestation statement for authenticator data authData and the clientDataHash computed in step 6.
11. If validation is successful, obtain a list of acceptable trust anchors (attestation root certificates or DAA root keys) for that attestation type and attestation statement format fmt, from a trusted source or from policy. For example, the FIDO Metadata Service [FIDOMetadataService] provides one way to access such information, using the AAGUID in authData.
12. Verify the trustworthiness of the attestation using the outputs of the verification process in step 10 as follows:
    + If self-attestation was used, check if self-attestation is acceptable under Relying Party policy.
    + If DAA was used, verify that the DAA key used is in the set of acceptable trust anchors obtained above.
    + Otherwise, use the X.509 certificates returned by the verification process to verify that the attestation public key correctly chains up to an acceptable root certificate.
13. If the attestation statement was correctly verified and found to be trustworthy, then register the new credential by associating the credential ID and credential public key found in authData with the Relying Party user on whose behalf the makeCredential() operation was requested.
14. If the attestation statement was correctly verified but could not be established to be trustworthy, the Relying Party SHOULD reject the registration operation. 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 5.3.2 Attestation Types). If doing so, the Relying Party is asserting there is no cryptographic proof that the Scoped Credential has been generated by a particular Authenticator model. See [FIDOSecRef] and [UAFProtocol] for a more detailed discussion.
15. If the attestation statement could not be correctly verified, the Relying Party MUST reject the registration operation.

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 MAY reject this operation, or it MAY decide to accept the registration while deleting the older registration.

6.2. Verifying an authentication assertion

When requested to perform authentication with an AuthenticationAssertion structure, the Relying Party MUST proceed as follows:

1. Using the credential identifier contained in the credential member of the AuthenticationAssertion structure, look up the corresponding credential public key as well as the Relying Party user for whom it is registered.
2. Using the procedure in 5.2.4 Verifying a signature, verify that signature is a valid signature over clientData and authenticatorData with the above public key.
3. If the above verification succeeds, authenticate the user looked up in step 1. Otherwise, reject the authentication request.

7. Defined Attestation Statement Formats

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

7.1. Attestation Format Identifiers

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

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

Unregistered attestation format identifiers SHOULD use reverse domain-name naming, using a domain name registered by the developer, in order to assure uniqueness of the identifier. All attestation format identifiers MUST be a maximum of 32 octets in length and MUST consist only of printable USASCII characters, i.e., VCHAR as defined in [RFC5234] (note: this means attestation format identifiers based on domain names MUST incorporate only LDH Labels [RFC5890]). Implementations MUST match WebAuthn attestation format identifiers in a case-insensitive fashion.

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

The following sections present a set of currently-defined and registered attestation statement formats and their identifiers. See the

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 MAY reject this operation, or it MAY decide to accept the registration while deleting the older registration.

6.2. Verifying an authentication assertion

When requested to perform authentication with an AuthenticationAssertion structure, the Relying Party MUST proceed as follows:

1. Using the credential identifier contained in the credential member of the AuthenticationAssertion structure, look up the corresponding credential public key as well as the Relying Party user for whom it is registered.
2. Using the procedure in 5.2.4 Verifying a signature, verify that signature is a valid signature over clientData and authenticatorData with the above public key.
3. If the above verification succeeds, authenticate the user looked up in step 1. Otherwise, reject the authentication request.

7. Defined Attestation Statement Formats

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

7.1. Attestation Format Identifiers

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

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

Unregistered attestation format identifiers SHOULD use reverse domain-name naming, using a domain name registered by the developer, in order to assure uniqueness of the identifier. All attestation format identifiers MUST be a maximum of 32 octets in length and MUST consist only of printable USASCII characters, i.e., VCHAR as defined in [RFC5234] (note: this means attestation format identifiers based on domain names MUST incorporate only LDH Labels [RFC5890]). Implementations MUST match WebAuthn attestation format identifiers in a case-insensitive fashion.

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

The following sections present a set of currently-defined and registered attestation statement formats and their identifiers. See the

WebAuthn Attestation Format Identifier Registry defined in [WebAuthn-Registries] for an up-to-date list of registered attestation statement formats.

7.2. Packed Attestation Statement Format

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

Attestation format identifier

packed

Attestation types supported

All

Syntax

A Packed Attestation statement is a CBOR map with the following fields:

alg

A text string containing the name of the algorithm used to generate the attestation signature according to [RFC7518] section 3.1. The following algorithms are supported:

1. "ES256", "ES384" and "ES512" [RFC7518]
2. "RS256", "RS384" and "RS512" [RFC7518]
3. "PS256", "PS384" and "PS512" [RFC7518]
4. "ED256" and "ED512" [FIDOEcdaaAlgorithm]

signature

A byte string containing the attestation signature.

x5c

A definite-length array of byte strings. The elements of the 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. This field is present only if Basic attestation or Privacy CA attestation is in use.

daaKey

A byte string containing the DAA root key. This field is present only if Direct Anonymous Attestation is in use.

Signing procedure

If Basic or Privacy CA attestation is in use, the authenticator produces signature by following the procedure in 5.2.3 Generating a signature with the given clientDataHash and authenticatorData as inputs, 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.

WebAuthn Attestation Format Identifier Registry defined in [WebAuthn-Registries] for an up-to-date list of registered attestation statement formats.

7.2. Packed Attestation Statement Format

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

Attestation format identifier

packed

Attestation types supported

All

Syntax

A Packed Attestation statement is a CBOR map with the following fields:

alg

A text string containing the name of the algorithm used to generate the attestation signature according to [RFC7518] section 3.1. The following algorithms are supported:

1. "ES256", "ES384" and "ES512" [RFC7518]
2. "RS256", "RS384" and "RS512" [RFC7518]
3. "PS256", "PS384" and "PS512" [RFC7518]
4. "ED256" and "ED512" [FIDOEcdaaAlgorithm]

sig

A byte string containing the attestation signature.

x5c

A definite-length array of byte strings. The elements of the 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. This field is present only if Basic attestation or Privacy CA attestation is in use.

daaKey

A byte string containing the DAA root key. This field is present only if Direct Anonymous Attestation is in use.

Signing procedure

If Basic or Privacy CA attestation is in use, the authenticator produces sig by following the procedure in 5.2.3 Generating a signature with the given clientDataHash and authenticatorData as inputs, using an attestation private key selected through an authenticator-specific mechanism. It sets x5c to the certificate chain of the attestation public key and alg to the algorithm of the attestation private key.

If DAA is in use, the authenticator produces signature by

following the procedure in 5.2.3 Generating a signature with the given clientDataHash and authenticatorData as inputs, using DAA-Sign with a DAA root key selected through an authenticator-specific mechanism (see [FIDOEcdaaAlgorithm]). It sets alg to the algorithm of the DAA root key and daaKey to the DAA root key.

If self attestation is in use, the authenticator produces signature by following the procedure in 5.2.3 Generating a signature with the given clientDataHash and authenticatorData as inputs, using the credential private key. It sets alg to the algorithm of the credential private key, and omits the other fields.

Verification procedure

If both x5c and daaKey are present, terminate this procedure with an error.

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

+ Follow the procedure in 5.2.4 Verifying a signature to verify that signature is a valid signature over the given authenticatorData and clientDataHash using the attestation public key in x5c with the algorithm specified in alg.
+ Verify that x5c meets the requirements in 7.2.1 Packed attestation statement certificate requirements.
+ If x5c contains an extension with OID 1 3 6 1 4 1 45724 1 1 4 (id-fido-gen-ce-aaguid) verify that the value of this extension matches the AAGUID in the claimed authenticatorData.
+ If successful, return attestation type Basic and trust path x5c.

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

+ Verify that alg is "ED256" or "ED512".
+ Follow the procedure in 5.2.4 Verifying a signature to verify that signature is a valid signature over the given authenticatorData and clientDataHash using DAA-Verify with daaKey (see [FIDOEcdaaAlgorithm]).
+ If successful, return attestation type DAA and trust path daaKey.

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

+ Validate that alg matches the algorithm of the credential private key in the claimed authenticatorData.
+ Construct attToBeSigned from the claimed authenticatorData and ClientData, and verify the signature using the credential public key.

If DAA is in use, the authenticator produces sig by following the procedure in 5.2.3 Generating a signature with the given clientDataHash and authenticatorData as inputs, using DAA-Sign with a DAA root key selected through an authenticator-specific mechanism (see [FIDOEcdaaAlgorithm]). It sets alg to the algorithm of the DAA root key and daaKey to the DAA root key.

If self attestation is in use, the authenticator produces sig by following the procedure in 5.2.3 Generating a signature with the given clientDataHash and authenticatorData as inputs, using the credential private key. It sets alg to the algorithm of the credential private key, and omits the other fields.

Verification procedure

If both x5c and daaKey are present, terminate this procedure with an error.

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

+ Follow the procedure in 5.2.4 Verifying a signature to verify that sig is a valid signature over the given authenticatorData and clientDataHash using the attestation public key in x5c with the algorithm specified in alg.
+ Verify that x5c meets the requirements in 7.2.1 Packed attestation statement certificate requirements.
+ If x5c contains an extension with OID 1 3 6 1 4 1 45724 1 1 4 (id-fido-gen-ce-aaguid) verify that the value of this extension matches the AAGUID in the claimed authenticatorData.
+ If successful, return attestation type Basic and trust path x5c.

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

+ Verify that alg is "ED256" or "ED512".
+ Follow the procedure in 5.2.4 Verifying a signature to verify that sig is a valid signature over the given authenticatorData and clientDataHash using DAA-Verify with daaKey (see [FIDOEcdaaAlgorithm]).
+ If successful, return attestation type DAA and trust path daaKey.

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

+ Validate that alg matches the algorithm of the credential private key in the claimed authenticatorData.

+ Follow the procedure in 5.2.4 Verifying a signature to verify that signature is a valid signature over the given authenticatorData and clientDataHash using the credential public key with alg.

+ If successful, return attestation type Self and empty trust path.

7.2.1. Packed attestation statement certificate requirements

The attestation certificate MUST have the following fields/extensions:

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

Subject-C

Country where the Authenticator vendor is incorporated

Subject-O

Legal name of the Authenticator vendor

Subject-OU

Authenticator Attestation

Subject-CN

No stipulation.

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

7.3. TPM Attestation Statement Format

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

Attestation format identifier

tpm

Attestation types supported

Privacy CA, DAA

Syntax

A TPM Attestation statement is a CBOR map with the following fields:

tpmVersion

A text string containing the version of the TPM

+ Follow the procedure in 5.2.4 Verifying a signature to verify that sig is a valid signature over the given authenticatorData and clientDataHash using the credential public key with alg.

+ If successful, return attestation type Self and empty trust path.

7.2.1. Packed attestation statement certificate requirements

The attestation certificate MUST have the following fields/extensions:

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

Subject-C

Country where the Authenticator vendor is incorporated

Subject-O

Legal name of the Authenticator vendor

Subject-OU

Authenticator Attestation

Subject-CN

No stipulation.

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

7.3. TPM Attestation Statement Format

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

Attestation format identifier

tpm

Attestation types supported

Privacy CA, DAA

Syntax

A TPM Attestation statement is a CBOR map with the following fields:

ver

A text string containing the version of the TPM

specification to which the signature conforms. Currently the only supported version is "2.0".

x5c

A definite-length array of byte strings. The elements of the array contain the AIK certificate used for the attestation, each encoded in X.509 format. The attestation certificate must be the first element in the array.

alg

A text string containing the name of the algorithm used to generate the attestation signature according to [RFC7518] section 3.1. The following algorithms are supported:

1. "ES256" [RFC7518]
2. "RS256" [RFC7518]
3. "PS256" [RFC7518]
4. "ED256" [FIDOEcdaaAlgorithm]
5. "ED512" [FIDOEcdaaAlgorithm]

signature

A byte string containing the attestation signature, in the form of a TPMT_SIGNATURE structure as specified in [TPMv2-Part2] section 11.3.4.

certifyInfo

A byte string containing the structure over which the attestation signature was computed. This is a TPMS_ATTEST structure as specified in [TPMv2-Part2] section 10.12.8.

publicArea

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

Signing procedure

Concatenate the given authenticatorData and clientDataHash as shown in 5.2.3 Generating a signature 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 publicArea field to the public area of the credential public key, the certifyInfo field to the output parameter of the same name, and the signature to the signature obtained from the above procedure.

Verification procedure

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

Concatenate the given authenticatorData and clientDataHash as

specification to which the signature conforms. Currently the only supported version is "2.0".

x5c

A definite-length array of byte strings. The elements of the array contain the AIK certificate used for the attestation, each encoded in X.509 format. The attestation certificate must be the first element in the array.

alg

A text string containing the name of the algorithm used to generate the attestation signature according to [RFC7518] section 3.1. The following algorithms are supported:

1. "ES256" [RFC7518]
2. "RS256" [RFC7518]
3. "PS256" [RFC7518]
4. "ED256" [FIDOEcdaaAlgorithm]
5. "ED512" [FIDOEcdaaAlgorithm]

sig

A byte string containing the attestation signature, in the form of a TPMT_SIGNATURE structure as specified in [TPMv2-Part2] section 11.3.4.

certInfo

A byte string containing the structure over which the attestation signature was computed. This is a TPMS_ATTEST structure as specified in [TPMv2-Part2] section 10.12.8.

pubArea

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

Signing procedure

Concatenate the given authenticatorData and clientDataHash as shown in 5.2.3 Generating a signature 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 public key specified by the parameters and unique fields of pubArea is identical to the public key contained in the attestation data inside the claimed authenticatorData

Concatenate the given authenticatorData and clientDataHash as shown in 5.2.3 Generating a signature to form attToBeSigned.

Validate that certifyInfo 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 publicArea, as computed using the algorithm in the nameAlg field of publicArea using the procedure specified in [TPMv2-Part1] section 16.

If both x5c and daaKey are present, terminate this procedure with an error.

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

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

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

+ Verify that alg is "ED256" or "ED512".
+ Perform DAA-Verify on signature to verify that it is a valid signature over certifyInfo (see [FIDOEcdaaAlgorithm]).
+ If successful, return attestation type DAA and trust path daaKey.

7.3.1. TPM attestation statement certificate requirements

TPM attestation certificate MUST have the following fields/extensions:

* Version must be set to 3.
* Subject field MUST be set to empty.
* The Subject Alternative Name extension must be set as defined in [TPMv2-EK-Profile] section 3.2.9.
* The Extended Key Usage extension MUST contain the "joint-iso-itu-t(2) internationalorganizations(23) 133 tcg-kp(8) tcg-kp-AIKCertificate(3)" OID.
* The Basic Constraints extension MUST have the CA component set to false.
* An Authority Information Access (AIA) extension with entry 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

shown in 5.2.3 Generating a signature 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 both x5c and daaKey are present, terminate this procedure with an error.

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

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

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

+ Verify that alg is "ED256" or "ED512".
+ Perform DAA-Verify on sig to verify that it is a valid signature over certInfo (see [FIDOEcdaaAlgorithm]).
+ If successful, return attestation type DAA and trust path daaKey.

7.3.1. TPM attestation statement certificate requirements

TPM attestation certificate MUST have the following fields/extensions:

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

7.4. Android Key Attestation Statement Format

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

Attestation format identifier

android-key

Attestation types supported

Basic

Syntax

An Android key attestation statement is a CBOR byte string containing the Android attestation statement, which is a series of DER encoded X.509 certificates. See the Android developer documentation.

Signing procedure

Concatenate the given authenticatorData and clientDataHash as shown in 5.2.3 Generating a signature to form attToBeSigned.

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

Verification procedure

Verification is performed as follows:

+ Verify that the public key in the first certificate in the series of certificates represented by the signature matches the credential public key in the attestation data field of the given authenticatorData.
+ Verify that in the attestation certificate extension data:
    o The value of the attestationChallenge field is identical to the concatenation of the claimed authenticatorData and clientDataHash.
    o The AuthorizationList.allApplications field is not present, since ScopedCredentials must be bound to the RP ID.
    o The value in the AuthorizationList.origin field is equal to KM_TAG_GENERATED.
    o The value in the AuthorizationList.purpose field is equal to KM_PURPOSE_SIGN.
+ If successful, return attestation type Basic with the trust path set to the entire attestation statement.

7.5. Android SafetyNet Attestation Statement Format

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

environment, but the authenticatorData is produced outside this environment. The Relying Party is expected to check that the contents of authenticatorData are consistent with the fields of the attestation certificate's extension data.

Attestation format identifier

android-key

Attestation types supported

Basic

Syntax

An Android key attestation statement is a CBOR byte string containing the Android attestation statement, which is a series of DER encoded X.509 certificates. See the Android developer documentation.

Signing procedure

Concatenate the given authenticatorData and clientDataHash as shown in 5.2.3 Generating a signature to form attToBeSigned.

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

Verification procedure

Verification is performed as follows:

+ Verify that the public key in the first certificate in the series of certificates represented by the signature matches the credential public key in the attestation data field of the given authenticatorData.
+ Verify that in the attestation certificate extension data:
    o The value of the attestationChallenge field is identical to the concatenation of the claimed authenticatorData and clientDataHash.
    o The AuthorizationList.allApplications field is not present, since ScopedCredentials must be bound to the RP ID.
    o The value in the AuthorizationList.origin field is equal to KM_TAG_GENERATED.
    o The value in the AuthorizationList.purpose field is equal to KM_PURPOSE_SIGN.
+ If successful, return attestation type Basic with the trust path set to the entire attestation statement.

7.5. Android SafetyNet Attestation Statement Format

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

Attestation format identifier

android-safetynet

Attestation types supported

Basic

Syntax

An Android Attestation statement is a CBOR map with the following fields:

version

A text string indicating the version number of Google Play Services responsible for providing the SafetyNet API.

safetyNetResponse

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

Signing procedure

Concatenate the given authenticatorData and clientDataHash as shown in 5.2.3 Generating a signature to form attToBeSigned.

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

Verification procedure

Verification is performed as follows:

+ Verify that safetyNetResponse is a valid SafetyNet response of version version.
+ Verify that the nonce in the safetyNetResponse is identical to the concatenation of the claimed authenticatorData and clientDataHash.
+ Verify that the attestation certificate is issued to the hostname "attest.android.com" (see SafetyNet online documentation).
+ Verify that the ctsProfileMatch attribute in the payload of safetyNetResponse 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 format identifier

fido-u2f

Attestation types supported

Attestation format identifier

android-safetynet

Attestation types supported

Basic

Syntax

An Android Attestation statement is a CBOR map with the following fields:

ver

A text string indicating the version number of Google Play Services responsible for providing the SafetyNet API.

response

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

Signing procedure

Concatenate the given authenticatorData and clientDataHash as shown in 5.2.3 Generating a signature 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 response is a valid SafetyNet response of version ver.
+ Verify that the nonce in the response is identical to the concatenation of the claimed authenticatorData and clientDataHash.
+ Verify that the attestation certificate is issued to the hostname "attest.android.com" (see SafetyNet online documentation).
+ Verify that the ctsProfileMatch attribute in the payload of response is true.
+ If successful, return attestation type Basic with the trust path set to the above attestation certificate.

7.6. FIDO U2F Attestation Statement Format

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

Attestation format identifier

fido-u2f

Attestation types supported

Basic

Syntax

A FIDO U2F attestation statement is a CBOR map with the following fields:

x5c

A byte string representing the U2F attestation certificate used for the attestation, encoded in X.509 format.

signature

A byte string containing the attestation signature.

Signing procedure

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

Concatenate the given authenticatorData and clientDataHash as shown in 5.2.3 Generating a signature to form attToBeSigned. Compute the SHA-256 hash of attToBeSigned and call the result tbsHash.

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 signature and set the attestation certificate of the attestation public key as x5c.

Verification procedure

Verification is performed as follows:

+ If x5c is not a certificate for an ECDSA public key over the P-256 curve, stop verification and return an error.
+ Concatenate the given authenticatorData and clientDataHash as shown in 5.2.3 Generating a signature to form attToBeSigned. Compute the SHA-256 hash of attToBeSigned and call the result tbsHash.
+ From the given authenticatorData, extract the claimed RP ID hash, the claimed credential ID and the claimed credential public key.
+ Generate the claimed to-be-signed data as specified in [FIDO-U2F-Message-Formats] section 4.3, with the application parameter set to the claimed RP ID hash, the challenge parameter set to tbsHash, the key handle parameter set to the claimed credential ID of the given credential, and the user public key parameter set to the claimed credential public key.
+ Verify that the signature is a valid ECDSA P-256 signature over the to-be-signed data constructed above.
+ If successful, return attestation type Basic with the trust path set to x5c.

8. WebAuthn Extensions

The mechanism for generating scoped credentials, as well as requesting

Basic

Syntax

A FIDO U2F attestation statement is a CBOR map with the following fields:

x5c

A byte string representing the U2F attestation certificate used for the attestation, encoded in X.509 format.

sig

A byte string containing the attestation signature.

Signing procedure

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

Concatenate the given authenticatorData and clientDataHash as shown in 5.2.3 Generating a signature to form attToBeSigned. Compute the SHA-256 hash of attToBeSigned and call the result tbsHash.

Generate a signature as specified in [FIDO-U2F-Message-Formats] section 4.3, with the application parameter set to the SHA-256 hash of the RP ID associated with the given credential, the challenge parameter set to tbsHash, and the key handle parameter set to the credential ID of the given credential. Set this as sig and set the attestation certificate of the attestation public key as x5c.

Verification procedure

Verification is performed as follows:

+ If x5c is not a certificate for an ECDSA public key over the P-256 curve, stop verification and return an error.
+ Concatenate the given authenticatorData and clientDataHash as shown in 5.2.3 Generating a signature to form attToBeSigned. Compute the SHA-256 hash of attToBeSigned and call the result tbsHash.
+ From the given authenticatorData, extract the claimed RP ID hash, the claimed credential ID and the claimed credential public key.
+ Generate the claimed to-be-signed data as specified in [FIDO-U2F-Message-Formats] section 4.3, with the application parameter set to the claimed RP ID hash, the challenge parameter set to tbsHash, the key handle parameter set to the claimed credential ID of the given credential, and the user public key parameter set to the claimed credential public key.
+ Verify that the sig is a valid ECDSA P-256 signature over the to-be-signed data constructed above.
+ If successful, return attestation type Basic with the trust path set to x5c.

8. WebAuthn Extensions

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

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

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

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

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

8.1. Extension Identifiers

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

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

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

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

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

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

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

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

8.1. Extension Identifiers

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

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

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

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

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

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

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

8.2. Defining extensions

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

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

8.3. Extending request parameters

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

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

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

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

Extensions that only affect client processing need not specify an authenticator argument. Extensions that affect authenticator processing MUST specify a method of computing the authenticator argument from the

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

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

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

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

8.2. Defining extensions

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

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

8.3. Extending request parameters

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

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

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

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

Extensions that only affect client processing need not specify an authenticator argument. Extensions that affect authenticator processing MUST specify a method of computing the authenticator argument from the

client argument. For extensions that do not require additional parameters, and are defined as taking a Boolean client argument set to true, this method SHOULD consist of passing an authenticator argument of true (CBOR major type 7, value 21).

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

8.4. Extending client processing

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

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

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

8.5. Extending authenticator processing

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

As specified in 5.2.1 Authenticator data, the authenticator data value of each processed extension is included in the extended data part of the authenticatorData. This part is a CBOR map, with extension identifiers as keys, and the authenticator data value of each extension as the value.

8.6. Example extension

This section is not normative.

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

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

```
var assertionPromise =
    credentials.getAssertion("SGFuIFNvbG8gc2hvdCBmaXJzdC4",
        {}, /* Empty filter */
        { 'webauthnExample_geo': true });
```

The extension defines the additional client data to be the client's

client argument. For extensions that do not require additional parameters, and are defined as taking a Boolean client argument set to true, this method SHOULD consist of passing an authenticator argument of true (CBOR major type 7, value 21).

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

8.4. Extending client processing

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

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

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

8.5. Extending authenticator processing

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

As specified in 5.2.1 Authenticator data, the authenticator data value of each processed extension is included in the extended data part of the authenticatorData. This part is a CBOR map, with extension identifiers as keys, and the authenticator data value of each extension as the value.

8.6. Example extension

This section is not normative.

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

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

var assertionPromise =
    credentials.getAssertion("SGFuIFNvbG8gc2hvdCBmaXJzdC4",
        {}, /* Empty filter */
        { 'webauthnExample_geo': true });

The extension defines the additional client data to be the client's

location, if known, as a GeoJSON [GeoJSON] point. The client constructs the following client data:

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

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

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

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

9. Pre-defined extensions

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

9.1. FIDO AppId

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

Extension identifier

fido_appid

Client argument

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

Client processing

If rpId is present, reject promise with a DOMException whose

location, if known, as a GeoJSON [GeoJSON] point. The client constructs the following client data:

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

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

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

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

9. Pre-defined extensions

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

9.1. FIDO AppId

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

Extension identifier

fido_appid

Client argument

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

Client processing

If rpId is present, reject promise with a DOMException whose

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

Authenticator argument

none

Authenticator processing

none

Authenticator data

none

9.2. Transaction authorization

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

Extension identifier

webauthn_txAuthSimple

Client argument

A single UTF-8 encoded string prompt.

Client processing

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

Authenticator argument

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

Authenticator processing

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

Authenticator data

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

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

Extension identifier

webauthn_txAuthGeneric

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

Authenticator argument

none

Authenticator processing

none

Authenticator data

none

9.2. Transaction authorization

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

Extension identifier

webauthn_txAuthSimple

Client argument

A single UTF-8 encoded string prompt.

Client processing

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

Authenticator argument

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

Authenticator processing

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

Authenticator data

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

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

Extension identifier

webauthn_txAuthGeneric

Client argument

A CBOR map with one pair of data items (CBOR tagged as 0xa1). The pair of data items consists of

1. one UTF-8 encoded string contentType, containing the MIME-Type of the content, e.g. "image/png"
2. and the content itself, encoded as CBOR byte array.

Client processing

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

Authenticator argument

The client argument encoded as a CBOR map.

Authenticator processing

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

Authenticator data

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

9.3. Authenticator Selection Extension

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

Extension identifier

webauthn_authnSel

Client argument

A sequence of AAGUIDs:

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

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

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

```
typedef BufferSource AAGUID;
```

Client processing

This extension can only be used during makeCredential(). If the

Client argument

A CBOR map with one pair of data items (CBOR tagged as 0xa1). The pair of data items consists of

1. one UTF-8 encoded string contentType, containing the MIME-Type of the content, e.g. "image/png"
2. and the content itself, encoded as CBOR byte array.

Client processing

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

Authenticator argument

The client argument encoded as a CBOR map.

Authenticator processing

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

Authenticator data

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

9.3. Authenticator Selection Extension

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

Extension identifier

webauthn_authnSel

Client argument

A sequence of AAGUIDs:

typedef sequence < AAGUID > AuthenticatorSelectionList;

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

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

typedef BufferSource AAGUID;

Client processing

This extension can only be used during makeCredential(). If the

client supports the Authenticator Selection Extension, it MUST use the first available authenticator whose AAGUID is present in the AuthenticatorSelectionList. If none of the available authenticators match a provided AAGUID, the client MUST select an authenticator from among the available authenticators to generate the credential.

Authenticator argument

There is no authenticator argument.

Authenticator processing

None.

9.4. SupportedExtensions Extension

Extension identifier

webauthn_exts

Client argument

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

Client processing

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

Authenticator argument

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

Authenticator processing

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

Authenticator data

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

9.5. User Verification Index (UVI) Extension

Extension identifier

webauthn_uvi

Client argument

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

Client processing

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

client supports the Authenticator Selection Extension, it MUST use the first available authenticator whose AAGUID is present in the AuthenticatorSelectionList. If none of the available authenticators match a provided AAGUID, the client MUST select an authenticator from among the available authenticators to generate the credential.

Authenticator argument

There is no authenticator argument.

Authenticator processing

None.

9.4. SupportedExtensions Extension

Extension identifier

webauthn_exts

Client argument

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

Client processing

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

Authenticator argument

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

Authenticator processing

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

Authenticator data

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

9.5. User Verification Index (UVI) Extension

Extension identifier

webauthn_uvi

Client argument

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

Client processing

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

Authenticator argument

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

Authenticator processing

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

Authenticator data

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

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

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

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

Example for authenticatorData containing one UVI extension

```
...                                          -- RP ID hash (32 bytes)
81                                           -- TUP and ED set
00 00 00 01                                  -- (initial) signature counter
...                                          -- all public key alg etc.
A1                                           -- extension: CBOR map of one elemen
t
    6C                                       -- Key 1: CBOR text string of 11 byt
es
        77 65 62 61 75 74 68 6E 5F 75 76 69 -- "webauthn_uvi" UTF-8 encoded stri
ng
    58 20                                    -- Value 1: CBOR byte string with 0x
20 bytes
        00 43 B8 E3 BE 27 95 8C              -- the UVI value itself
        28 D5 74 BF 46 8A 85 CF
        46 9A 14 F0 E5 16 69 31
        DA 4B CF FF C1 BB 11 32
        82
```

9.6. Location Extension

Extension identifier

Authenticator argument

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

Authenticator processing

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

Authenticator data

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

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

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

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

Example for authenticatorData containing one UVI extension

```
...                                          -- RP ID hash (32 bytes)
81                                           -- TUP and ED set
00 00 00 01                                  -- (initial) signature counter
...                                          -- all public key alg etc.
A1                                           -- extension: CBOR map of one elemen
t
    6C                                       -- Key 1: CBOR text string of 11 byt
es
        77 65 62 61 75 74 68 6E 5F 75 76 69 -- "webauthn_uvi" UTF-8 encoded stri
ng
    58 20                                    -- Value 1: CBOR byte string with 0x
20 bytes
        00 43 B8 E3 BE 27 95 8C              -- the UVI value itself
        28 D5 74 BF 46 8A 85 CF
        46 9A 14 F0 E5 16 69 31
        DA 4B CF FF C1 BB 11 32
        82
```

9.6. Location Extension

Extension identifier

webauthn_loc

Client argument

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

Client processing

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

Authenticator argument

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

Authenticator processing

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

Authenticator data

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

```
...                                          -- RP ID hash (32 bytes)
81                                           -- TUP and ED set
00 00 00 01                                  -- (initial) signature counter
...                                          -- all public key alg etc.
A1                                           -- extension: CBOR map of one elemen
t
    6C                                       -- Value 1: CBOR text string of 11 b
ytes
        77 65 62 61 75 74 68 6E 5F 6C 6F 63 -- "webauthn_loc" UTF-8 encoded stri
ng
    86                                       -- Value 2: array of 6 elements
        68                -- Element 1:  CBOR text string of 8 bytes
           6C 61 74 69 74 75 64 65          -- "latitude" UTF-8 encoded string
        FB ...                -- Element 2:  Latitude as CBOR encoded double-p
recision float
        69                -- Element 3:  CBOR text string of 9 bytes
           6C 6F 6E 67 69 74 75 64 65       -- "longitude" UTF-8 encoded string
        FB ...                -- Element 4:  Longitude as CBOR encoded double-
precision float
        68                -- Element 5:  CBOR text string of 8 bytes
          61 6C 74 69 74 75 64 65           -- "altitude" UTF-8 encoded string
        FB ...                -- Element 6:  Altitude as CBOR encoded double-p
recision float
```


webauthn_loc

Client argument

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

Client processing

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

Authenticator argument

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

Authenticator processing

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

Authenticator data

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

```
...                                          -- RP ID hash (32 bytes)
81                                           -- TUP and ED set
00 00 00 01                                  -- (initial) signature counter
...                                          -- all public key alg etc.
A1                                           -- extension: CBOR map of one elemen
t
    6C                                       -- Value 1: CBOR text string of 11 b
ytes
        77 65 62 61 75 74 68 6E 5F 6C 6F 63 -- "webauthn_loc" UTF-8 encoded stri
ng
    86                                       -- Value 2: array of 6 elements
        68                -- Element 1:  CBOR text string of 8 bytes
           6C 61 74 69 74 75 64 65          -- "latitude" UTF-8 encoded string
        FB ...                -- Element 2:  Latitude as CBOR encoded double-p
recision float
        69                -- Element 3:  CBOR text string of 9 bytes
           6C 6F 6E 67 69 74 75 64 65       -- "longitude" UTF-8 encoded string
        FB ...                -- Element 4:  Longitude as CBOR encoded double-
precision float
        68                -- Element 5:  CBOR text string of 8 bytes
          61 6C 74 69 74 75 64 65           -- "altitude" UTF-8 encoded string
        FB ...                -- Element 6:  Altitude as CBOR encoded double-p
recision float
```

9.7. User Verification Mode (UVM) Extension

Extension identifier

webauthn_uvm

Client argument

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

Client processing

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

Authenticator argument

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

Authenticator processing

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

Authenticator data

Authenticators can report up to 3 different user verification methods (factors) used in a single authentication instance. To accommodate this possibility the UVM is encoded as CBOR array (major type 4) with a maximum allowed length of 3 -

+ Type 0x81 - only 1 factor was used for authentication.
+ Type 0x82 - 2 factors were used.
+ Type 0x83 - 3 or more factors were used.

Each data item is in turn a CBOR array of length 3 (type 0x83) with the following data items:

+ Data Item 1 - User Verification Method. This is the authentication method/factor used by the authenticator to verify the user. Available values are defined in [FIDOReg], "User Verification Methods" section. It is encoded as a CBOR unsigned integer (Major type 0).
+ Data Item 2 - Key Protection Type. This is the method used by the authenticator to protect the FIDO registration private key material. Available values are defined in [FIDOReg], "Key Protection Types" section. It is encoded as a CBOR 2 byte unsigned short (Major type 0).
+ Data Item 3 - Matcher Protection Type. This is the method used by the authenticator to protect the matcher that performs user verification. Available values are defined in [FIDOReg], "Matcher Protection Types" section. It is encoded as a CBOR 2 byte unsigned short (Major type 0).

This is repeated for each factor used in the authentication instance.

If >3 factors can be used in an authentication instance the

9.7. User Verification Mode (UVM) Extension

Extension identifier

webauthn_uvm

Client argument

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

Client processing

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

Authenticator argument

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

Authenticator processing

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

Authenticator data

Authenticators can report up to 3 different user verification methods (factors) used in a single authentication instance. To accommodate this possibility the UVM is encoded as CBOR array (major type 4) with a maximum allowed length of 3 -

+ Type 0x81 - only 1 factor was used for authentication.
+ Type 0x82 - 2 factors were used.
+ Type 0x83 - 3 or more factors were used.

Each data item is in turn a CBOR array of length 3 (type 0x83) with the following data items:

+ Data Item 1 - User Verification Method. This is the authentication method/factor used by the authenticator to verify the user. Available values are defined in [FIDOReg], "User Verification Methods" section. It is encoded as a CBOR unsigned integer (Major type 0).
+ Data Item 2 - Key Protection Type. This is the method used by the authenticator to protect the FIDO registration private key material. Available values are defined in [FIDOReg], "Key Protection Types" section. It is encoded as a CBOR 2 byte unsigned short (Major type 0).
+ Data Item 3 - Matcher Protection Type. This is the method used by the authenticator to protect the matcher that performs user verification. Available values are defined in [FIDOReg], "Matcher Protection Types" section. It is encoded as a CBOR 2 byte unsigned short (Major type 0).

This is repeated for each factor used in the authentication instance.

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.

Servers supporting the UVM extension MUST support a length up to 36 bytes for a 3 factor maximum UVM value.

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

```
...                    -- RP ID hash (32 bytes)
81                     -- TUP and ED set
00 00 00 01            -- (initial) signature counter
...                    -- all public key alg etc.
A1                     -- extension: CBOR map of one element
    6C                 -- Key 1: CBOR text string of 12 bytes
        77 65 62 61 75 74 68 6E 2E 75 76 6d -- "webauthn_uvm" UTF-8 encoded stri
ng
    82                 -- Value 1: CBOR array of length 2 indicating two factor
usage
        83              -- Item 1: CBOR array of length 3
            02           -- Subitem 1: CBOR integer for User Verification Method
 Fingerprint
            04           -- Subitem 2: CBOR short for Key Protection Type TEE
            02           -- Subitem 3: CBOR short for Matcher Protection Type TE
E
        83              -- Item 2: CBOR array of length 3
            04           -- Subitem 1: CBOR integer for User Verification Method
 Passcode
            01           -- Subitem 2: CBOR short for Key Protection Type Softwa
re
            01           -- Subitem 3: CBOR short for Matcher Protection Type So
ftware
```

10. IANA Considerations

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

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

```
Algorithm Name                    "S256"
Algorithm Description             The SHA256 hash algorithm.
Algorithm Usage Location(s)       "alg", i.e., used with JWS.
JOSE Implementation Requirements Optional+
Change Controller                 FIDO Alliance
Specification Documents           [FIPS-180-4]
Algorithm Analysis Document(s)    [SP800-107r1]

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

Algorithm Name                    "S512"
Algorithm Description             The SHA512 hash algorithm.
Algorithm Usage Location(s)       "alg", i.e., used with JWS.
JOSE Implementation Requirements Optional+
```


authenticator vendor must select the 3 factors it believes will be most relevant to the Server to include in the UVM.

Servers supporting the UVM extension MUST support a length up to 36 bytes for a 3 factor maximum UVM value.

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

```
...                    -- RP ID hash (32 bytes)
81                     -- TUP and ED set
00 00 00 01            -- (initial) signature counter
...                    -- all public key alg etc.
A1                     -- extension: CBOR map of one element
    6C                 -- Key 1: CBOR text string of 12 bytes
        77 65 62 61 75 74 68 6E 2E 75 76 6d -- "webauthn_uvm" UTF-8 encoded stri
ng
    82                 -- Value 1: CBOR array of length 2 indicating two factor
usage
        83              -- Item 1: CBOR array of length 3
            02           -- Subitem 1: CBOR integer for User Verification Method
 Fingerprint
            04           -- Subitem 2: CBOR short for Key Protection Type TEE
            02           -- Subitem 3: CBOR short for Matcher Protection Type TE
E
        83              -- Item 2: CBOR array of length 3
            04           -- Subitem 1: CBOR integer for User Verification Method
 Passcode
            01           -- Subitem 2: CBOR short for Key Protection Type Softwa
re
            01           -- Subitem 3: CBOR short for Matcher Protection Type So
ftware
```

10. IANA Considerations

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

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

```
Algorithm Name                    "S256"
Algorithm Description             The SHA256 hash algorithm.
Algorithm Usage Location(s)       "alg", i.e., used with JWS.
JOSE Implementation Requirements Optional+
Change Controller                 FIDO Alliance
Specification Documents           [FIPS-180-4]
Algorithm Analysis Document(s)    [SP800-107r1]

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

Algorithm Name                    "S512"
Algorithm Description             The SHA512 hash algorithm.
Algorithm Usage Location(s)       "alg", i.e., used with JWS.
JOSE Implementation Requirements Optional+
```

Change Controller	FIDO Alliance
Specification Documents	[FIPS-180-4]
Algorithm Analysis Document(s)	[SP800-107r1]

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

11. Sample scenarios

This section is not normative.

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

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

11.1. Registration

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

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

Change Controller	FIDO Alliance
Specification Documents	[FIPS-180-4]
Algorithm Analysis Document(s)	[SP800-107r1]

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

11. Sample scenarios

This section is not normative.

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

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

11.1. Registration

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

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

also storing a friendly name for later use.
+ The script may store data such as the credential ID in local storage, to improve future UX by narrowing the choice of credential for the user.

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

```
var webauthnAPI = navigator.authentication;

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

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

// This Relying Party will accept either an ES256 or RS256 credential, but
// prefers an ES256 credential.
var cryptoParams = [
    {
        type: "ScopedCred",
        algorithm: "ES256"
    },
    {
        type: "ScopedCred",
        algorithm: "RS256"
    }
];
var challenge = "Y2xpbWIgYSBtb3VudGFpbg";
var options = { timeoutSeconds: 300,  // 5 minutes
                excludeList: [],      // No excludeList
                extensions: {"webauthn.location": true}  // Include location inf
ormation
                                               // in attestation
};

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

11.2. Authentication

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

1. The user visits example.com, which serves up a script.
2. The script asks the client platform for an Authentication Assertion, providing as much information as possible to narrow the choice of acceptable credentials for the user. This may be obtained from the data that was stored locally after registration, or by other means such as prompting the user for a username.
3. The Relying Party script runs one of the code snippets below.
4. The client platform searches for and locates the authenticator.
5. The client platform connects to the authenticator, performing any pairing actions if necessary.

also storing a friendly name for later use.
+ The script may store data such as the credential ID in local storage, to improve future UX by narrowing the choice of credential for the user.

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

```
var webauthnAPI = navigator.authentication;

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

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

// This Relying Party will accept either an ES256 or RS256 credential, but
// prefers an ES256 credential.
var cryptoParams = [
    {
        type: "ScopedCred",
        algorithm: "ES256"
    },
    {
        type: "ScopedCred",
        algorithm: "RS256"
    }
];
var challenge = "Y2xpbWIgYSBtb3VudGFpbg";
var options = { timeoutSeconds: 300,  // 5 minutes
                excludeList: [],      // No excludeList
                extensions: {"webauthn.location": true}  // Include location inf
ormation
                                               // in attestation
};

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

11.2. Authentication

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

1. The user visits example.com, which serves up a script.
2. The script asks the client platform for an Authentication Assertion, providing as much information as possible to narrow the choice of acceptable credentials for the user. This may be obtained from the data that was stored locally after registration, or by other means such as prompting the user for a username.
3. The Relying Party script runs one of the code snippets below.
4. The client platform searches for and locates the authenticator.
5. The client platform connects to the authenticator, performing any pairing actions if necessary.

6. The authenticator presents the user with a notification that their attention is required. On opening the notification, the user is shown a friendly selection menu of acceptable credentials using the account information provided when creating the credentials, along with some information on the origin that is requesting these keys.
7. The authenticator obtains a biometric or other authorization gesture from the user.
8. The authenticator returns a response to the client platform, which in turn returns a response to the Relying Party script. If the user declined to select a credential or provide an authorization, an appropriate error is returned.
9. If an assertion was successfully generated and returned,
   + The script sends the assertion to the server.
   + The server examines the assertion, extracts the credential ID, looks up the registered credential public key it is database, and verifies the assertion's authentication signature. If valid, it looks up the identity associated with the assertion's credential ID; that identity is now authenticated. If the credential ID is not recognized by the server (e.g., it has been deregistered due to inactivity) then the authentication has failed; each Relying Party will handle this in its own way.
   + The server now does whatever it would otherwise do upon successful authentication -- return a success page, set authentication cookies, etc.

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

```
var webauthnAPI = navigator.authentication;

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

var challenge = "Y2xpbWIgYSBtb3VudGFpbg";
var options = {
                timeoutSeconds = 300,  // 5 minutes
                allowList: [{ type: "ScopedCred" }]
              };

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

On the other hand, if the Relying Party script has some hints to help it narrow the list of credentials, then the sample code for performing such an authentication might look like the following. Note that this sample also demonstrates how to use the extension for transaction authorization.

```
var webauthnAPI = navigator.authentication;

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

var challenge = "Y2xpbWIgYSBtb3VudGFpbg";
var acceptableCredential1 = {
    type: "ScopedCred",
    id: "ISEhISEhIWhpIHRoZXJlISEhISEhIQo="
};
var acceptableCredential2 = {
```


6. The authenticator presents the user with a notification that their attention is required. On opening the notification, the user is shown a friendly selection menu of acceptable credentials using the account information provided when creating the credentials, along with some information on the origin that is requesting these keys.
7. The authenticator obtains a biometric or other authorization gesture from the user.
8. The authenticator returns a response to the client platform, which in turn returns a response to the Relying Party script. If the user declined to select a credential or provide an authorization, an appropriate error is returned.
9. If an assertion was successfully generated and returned,
   + The script sends the assertion to the server.
   + The server examines the assertion, extracts the credential ID, looks up the registered credential public key it is database, and verifies the assertion's authentication signature. If valid, it looks up the identity associated with the assertion's credential ID; that identity is now authenticated. If the credential ID is not recognized by the server (e.g., it has been deregistered due to inactivity) then the authentication has failed; each Relying Party will handle this in its own way.
   + The server now does whatever it would otherwise do upon successful authentication -- return a success page, set authentication cookies, etc.

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

```
var webauthnAPI = navigator.authentication;

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

var challenge = "Y2xpbWIgYSBtb3VudGFpbg";
var options = {
                timeoutSeconds = 300,  // 5 minutes
                allowList: [{ type: "ScopedCred" }]
              };

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

On the other hand, if the Relying Party script has some hints to help it narrow the list of credentials, then the sample code for performing such an authentication might look like the following. Note that this sample also demonstrates how to use the extension for transaction authorization.

```
var webauthnAPI = navigator.authentication;

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

var challenge = "Y2xpbWIgYSBtb3VudGFpbg";
var acceptableCredential1 = {
    type: "ScopedCred",
    id: "ISEhISEhIWhpIHRoZXJlISEhISEhIQo="
};
var acceptableCredential2 = {
```

```
    type: "ScopedCred",
    id: "cm9zZXMgYXJlIHJlZCwgdmlvbGV0cyBhcmUgYmx1ZQo="
};

var options = {
                timeoutSeconds: 300,  // 5 minutes
                allowList: [acceptableCredential1, acceptableCredential2];
                extensions: { 'webauthn.txauth.simple':
                   "Wave your hands in the air like you just don't care" };
              };

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

11.3. Decommissioning

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

* Possibility #1 -- user reports the credential as lost.
    + User goes to server.example.net, authenticates and follows a link to report a lost/stolen device.
    + Server returns a page showing the list of registered credentials with friendly names as configured during registration.
    + User selects a credential and the server deletes it from its database.
    + In future, the Relying Party script does not specify this credential in any list of acceptable credentials, and assertions signed by this credential are rejected.
* Possibility #2 -- server deregisters the credential due to inactivity.
    + Server deletes credential from its database during maintenance activity.
    + In the future, the Relying Party script does not specify this credential in any list of acceptable credentials, and assertions signed by this credential are rejected.
* Possibility #3 -- user deletes the credential from the device.
    + User employs a device-specific method (e.g., device settings UI) to delete a credential from their device.
    + From this point on, this credential will not appear in any selection prompts, and no assertions can be generated with it.
    + Sometime later, the server deregisters this credential due to inactivity.

12. Acknowledgements

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

Index

Terms defined by this specification


```
    type: "ScopedCred",
    id: "cm9zZXMgYXJlIHJlZCwgdmlvbGV0cyBhcmUgYmx1ZQo="
};

var options = {
                timeoutSeconds: 300,  // 5 minutes
                allowList: [acceptableCredential1, acceptableCredential2];
                extensions: { 'webauthn.txauth.simple':
                   "Wave your hands in the air like you just don't care" };
              };

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

11.3. Decommissioning

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

* Possibility #1 -- user reports the credential as lost.
    + User goes to server.example.net, authenticates and follows a link to report a lost/stolen device.
    + Server returns a page showing the list of registered credentials with friendly names as configured during registration.
    + User selects a credential and the server deletes it from its database.
    + In future, the Relying Party script does not specify this credential in any list of acceptable credentials, and assertions signed by this credential are rejected.
* Possibility #2 -- server deregisters the credential due to inactivity.
    + Server deletes credential from its database during maintenance activity.
    + In the future, the Relying Party script does not specify this credential in any list of acceptable credentials, and assertions signed by this credential are rejected.
* Possibility #3 -- user deletes the credential from the device.
    + User employs a device-specific method (e.g., device settings UI) to delete a credential from their device.
    + From this point on, this credential will not appear in any selection prompts, and no assertions can be generated with it.
    + Sometime later, the server deregisters this credential due to inactivity.

12. Acknowledgements

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

Index

Terms defined by this specification

* AAGUID, in 9.3
* Account, in 4.3

/Users/jehodges/Documents/work/standards/W3C/webauthn/index-vgb-u2f-attestation-0d0fcea.txt, Top line: 3402

/Users/jehodges/Documents/work/standards/W3C/webauthn/index-vgb-u2f-attestation-dc90eab.txt, Top line: 3393

/Users/jehodges/Documents/work/standards/W3C/webauthn/index-vgb-u2f-attestation-dc90eab.txt, Top line: 3454

* clientData
    + attribute for ScopedCredentialInfo, in 4.2
    + dfn for ScopedCredentialInfo, in 4.2
    + attribute for AuthenticationAssertion, in 4.6
    + dfn for AuthenticationAssertion, in 4.6
* ClientData, in 4.9.1
* clientDataHash, in 4.9.1
* clientDataJSON, in 4.9.1
* Conforming User Agent, in 3
* content, in 9.2
* contentType, in 9.2
* credential
    + attribute for AuthenticationAssertion, in 4.6
    + dfn for AuthenticationAssertion, in 4.6
* credential key pair, in 3
* credential private key, in 3
* Credential Public Key, in 3
* cross-platform, in 4.5.1
* "cross-platform", in 4.5.1
* cross-platform attached, in 4.5.1
* cross-platform attachment, in 4.5.1
* cryptoParameters, in 4.1.1
* Direct Anonymous Attestation, in 5.3.2
* displayName
    + dict-member for Account, in 4.3
    + dfn for Account, in 4.3
* DOMException, in 2.1
* excludeList
    + dict-member for ScopedCredentialOptions, in 4.5
    + dfn for ScopedCredentialOptions, in 4.5
* extension identifier, in 8.1
* extensions
    + dict-member for ScopedCredentialOptions, in 4.5
    + dfn for ScopedCredentialOptions, in 4.5
    + dict-member for AssertionOptions, in 4.7
    + dfn for AssertionOptions, in 4.7
    + dict-member for ClientData, in 4.9.1
    + dfn for ClientData, in 4.9.1
* ExternalTransport, in 4.9.5
* getAssertion(assertionChallenge), in 4.1.2
* getAssertion(assertionChallenge, options), in 4.1.2
* hashAlg
    + dict-member for ClientData, in 4.9.1
    + dfn for ClientData, in 4.9.1
* id
    + dict-member for Account, in 4.3
    + dfn for Account, in 4.3
    + attribute for ScopedCredential, in 4.9.3
    + dfn for ScopedCredential, in 4.9.3
    + dict-member for ScopedCredentialDescriptor, in 4.9.4
    + dfn for ScopedCredentialDescriptor, in 4.9.4
* imageURL
    + dict-member for Account, in 4.3
    + dfn for Account, in 4.3
* makeCredential(accountInformation, cryptoParameters, attestationChallenge), in 4.1.1
* makeCredential(accountInformation, cryptoParameters, attestationChallenge, options), in 4.1.1
* name
    + dict-member for Account, in 4.3
    + dfn for Account, in 4.3
* nfc

```
         + enum-value for Transport, in 4.9.5
         + dfn for Transport, in 4.9.5
     * "nfc", in 4.9.5
     * options
         + dfn for makeCredential(), in 4.1.1
         + dfn for getAssertion(), in 4.1.2
     * origin
         + dict-member for ClientData, in 4.9.1
         + dfn for ClientData, in 4.9.1
     * platform, in 4.5.1
     * "platform", in 4.5.1
     * platform attachment, in 4.5.1
     * platform authenticators, in 4.5.1
     * Privacy CA, in 5.3.2
     * Promises, in 2.1
     * Registration, in 3
     * Relying Party, in 3
     * Relying Party Identifier, in 3
     * roaming authenticators, in 4.5.1
     * rpDisplayName
         + dict-member for Account, in 4.3
         + dfn for Account, in 4.3
     * rpId
         + dict-member for ScopedCredentialOptions, in 4.5
         + dfn for ScopedCredentialOptions, in 4.5
         + dict-member for AssertionOptions, in 4.7
         + dfn for AssertionOptions, in 4.7
     * RP ID, in 3
     * ScopedCred
         + enum-value for ScopedCredentialType, in 4.9.2
         + dfn for ScopedCredentialType, in 4.9.2
     * "ScopedCred", in 4.9.2
     * Scoped Credential, in 3
     * ScopedCredential, in 4.9.3
     * ScopedCredentialDescriptor, in 4.9.4
     * ScopedCredentialInfo, in 4.2
     * ScopedCredentialOptions, in 4.5
     * ScopedCredentialParameters, in 4.4
     * ScopedCredentialType, in 4.9.2
     * secure contexts, in 4
     * Self Attestation, in 5.3.2
     * signature
         + attribute for AuthenticationAssertion, in 4.6
         + dfn for AuthenticationAssertion, in 4.6
     * timeoutSeconds
         + dict-member for ScopedCredentialOptions, in 4.5
         + dfn for ScopedCredentialOptions, in 4.5
         + dict-member for AssertionOptions, in 4.7
         + dfn for AssertionOptions, in 4.7
     * tokenBinding
         + dict-member for ClientData, in 4.9.1
         + dfn for ClientData, in 4.9.1
     * Transport, in 4.9.5
     * transports, in 4.9.4
     * type
         + dict-member for ScopedCredentialParameters, in 4.4
         + dfn for ScopedCredentialParameters, in 4.4
         + attribute for ScopedCredential, in 4.9.3
         + dfn for ScopedCredential, in 4.9.3
         + dict-member for ScopedCredentialDescriptor, in 4.9.4
         + dfn for ScopedCredentialDescriptor, in 4.9.4
     * "usb", in 4.9.5
```



```
         + enum-value for Transport, in 4.9.5
         + dfn for Transport, in 4.9.5
     * "nfc", in 4.9.5
     * options
         + dfn for makeCredential(), in 4.1.1
         + dfn for getAssertion(), in 4.1.2
     * origin
         + dict-member for ClientData, in 4.9.1
         + dfn for ClientData, in 4.9.1
     * platform, in 4.5.1
     * "platform", in 4.5.1
     * platform attachment, in 4.5.1
     * platform authenticators, in 4.5.1
     * Privacy CA, in 5.3.2
     * Promises, in 2.1
     * Registration, in 3
     * Relying Party, in 3
     * Relying Party Identifier, in 3
     * roaming authenticators, in 4.5.1
     * rpDisplayName
         + dict-member for Account, in 4.3
         + dfn for Account, in 4.3
     * rpId
         + dict-member for ScopedCredentialOptions, in 4.5
         + dfn for ScopedCredentialOptions, in 4.5
         + dict-member for AssertionOptions, in 4.7
         + dfn for AssertionOptions, in 4.7
     * RP ID, in 3
     * ScopedCred
         + enum-value for ScopedCredentialType, in 4.9.2
         + dfn for ScopedCredentialType, in 4.9.2
     * "ScopedCred", in 4.9.2
     * Scoped Credential, in 3
     * ScopedCredential, in 4.9.3
     * ScopedCredentialDescriptor, in 4.9.4
     * ScopedCredentialInfo, in 4.2
     * ScopedCredentialOptions, in 4.5
     * ScopedCredentialParameters, in 4.4
     * ScopedCredentialType, in 4.9.2
     * secure contexts, in 4
     * Self Attestation, in 5.3.2
     * signature
         + attribute for AuthenticationAssertion, in 4.6
         + dfn for AuthenticationAssertion, in 4.6
     * timeoutSeconds
         + dict-member for ScopedCredentialOptions, in 4.5
         + dfn for ScopedCredentialOptions, in 4.5
         + dict-member for AssertionOptions, in 4.7
         + dfn for AssertionOptions, in 4.7
     * tokenBinding
         + dict-member for ClientData, in 4.9.1
         + dfn for ClientData, in 4.9.1
     * Transport, in 4.9.5
     * transports, in 4.9.4
     * type
         + dict-member for ScopedCredentialParameters, in 4.4
         + dfn for ScopedCredentialParameters, in 4.4
         + attribute for ScopedCredential, in 4.9.3
         + dfn for ScopedCredential, in 4.9.3
         + dict-member for ScopedCredentialDescriptor, in 4.9.4
         + dfn for ScopedCredentialDescriptor, in 4.9.4
     * "usb", in 4.9.5
```

* usb
    + enum-value for Transport, in 4.9.5
    + dfn for Transport, in 4.9.5
* User Consent, in 3
* User Verification, in 3
* WebAuthentication, in 4.1
* Web Authentication API, in 4
* WebAuthn Client, in 3

Terms defined by reference

* [HTML] defines the following terms:
    + Navigator
* [HTML51] defines the following terms:
    + current settings object
    + navigator
    + opaque origin
    + origin
    + relaxing the same-origin restriction
* [WebCryptoAPI] defines the following terms:
    + normalizing an algorithm
* [WebIDL] defines the following terms:
    + BufferSource
    + SecureContext
    + present

References

Normative References

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

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

[FIDOReg]
R. Lindemann; D. Baghdasaryan; B. Hill. FIDO UAF Registry of Predefined Values. FIDO Alliance Proposed Standard. URL: https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-reg-v1.0-ps-20141208.html

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

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

[HTML51]
Steve Faulkner; et al. HTML 5.1. URL: https://w3c.github.io/html/

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

[RFC2119]
S. Bradner. Key words for use in RFCs to Indicate Requirement

* usb
    + enum-value for Transport, in 4.9.5
    + dfn for Transport, in 4.9.5
* User Consent, in 3
* User Verification, in 3
* WebAuthentication, in 4.1
* Web Authentication API, in 4
* WebAuthn Client, in 3

Terms defined by reference

* [HTML] defines the following terms:
    + Navigator
* [HTML51] defines the following terms:
    + current settings object
    + navigator
    + opaque origin
    + origin
    + relaxing the same-origin restriction
* [WebCryptoAPI] defines the following terms:
    + normalizing an algorithm
* [WebIDL] defines the following terms:
    + BufferSource
    + SecureContext
    + present

References

Normative References

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

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

[FIDOReg]
R. Lindemann; D. Baghdasaryan; B. Hill. FIDO UAF Registry of Predefined Values. FIDO Alliance Proposed Standard. URL: https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-reg-v1.0-ps-20141208.html

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

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

[HTML51]
Steve Faulkner; et al. HTML 5.1. URL: https://w3c.github.io/html/

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

[RFC2119]
S. Bradner. Key words for use in RFCs to Indicate Requirement

Levels. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

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

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

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

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

[WebCryptoAPI]
Mark Watson. Web Cryptography API. URL: https://w3c.github.io/webcrypto/Overview.html

[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-se

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

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

[WebCryptoAPI]
Mark Watson. Web Cryptography API. URL: https://w3c.github.io/webcrypto/Overview.html

[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-se

curity-ref-v1.0-ps-20141208.html

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

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

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

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

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

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

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

[SECURE-CONTEXTS]
Mike West. Secure Contexts. URL: https://w3c.github.io/webappsec-secure-contexts/

[SP800-107r1]
Quynh Dang. NIST Special Publication 800-107: Recommendation for Applications Using Approved Hash Algorithms. August 2012. URL: http://csrc.nist.gov/publications/nistpubs/800-107-rev1/sp800-10 7-rev1.pdf

[TPMv2-EK-Profile]
TCG EK Credential Profile for TPM Family 2.0. URL: http://www.trustedcomputinggroup.org/wp-content/uploads/Credenti al_Profile_EK_V2.0_R14_published.pdf

[TPMv2-Part1]
Trusted Platform Module Library, Part 1: Architecture. URL: http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-1-Architecture-01.16-1.pdf

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

[TPMv2-Part3]
Trusted Platform Module Library, Part 3: Commands. URL:

curity-ref-v1.0-ps-20141208.html

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

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

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

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

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

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

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

[SECURE-CONTEXTS]
Mike West. Secure Contexts. URL: https://w3c.github.io/webappsec-secure-contexts/

[SP800-107r1]
Quynh Dang. NIST Special Publication 800-107: Recommendation for Applications Using Approved Hash Algorithms. August 2012. URL: http://csrc.nist.gov/publications/nistpubs/800-107-rev1/sp800-10 7-rev1.pdf

[TPMv2-EK-Profile]
TCG EK Credential Profile for TPM Family 2.0. URL: http://www.trustedcomputinggroup.org/wp-content/uploads/Credenti al_Profile_EK_V2.0_R14_published.pdf

[TPMv2-Part1]
Trusted Platform Module Library, Part 1: Architecture. URL: http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-1-Architecture-01.16-1.pdf

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

[TPMv2-Part3]
Trusted Platform Module Library, Part 3: Commands. URL:

http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-3-Commands-01.16-1.pdf

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

[WebAuthn-Registries]
Jeff Hodges. Registries for Web Authentication (WebAuthn). June 2016. Active Internet-Draft. URL: https://xml2rfc.tools.ietf.org/cgi-bin/xml2rfc.cgi?modeAsFormat=html/ascii&url=https://raw.githubusercontent.com/w3c/webauthn/master/draft-hodges-webauthn-registries.xml#4692311055485507473 2

IDL Index

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

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

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

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

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

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

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


http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-3-Commands-01.16-1.pdf

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

[WebAuthn-Registries]
Jeff Hodges. Registries for Web Authentication (WebAuthn). June 2016. Active Internet-Draft. URL: https://xml2rfc.tools.ietf.org/cgi-bin/xml2rfc.cgi?modeAsFormat=html/ascii&url=https://raw.githubusercontent.com/w3c/webauthn/master/draft-hodges-webauthn-registries.xml#4692311055485507473 2

IDL Index

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

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

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

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

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

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

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

```
};

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

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

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

dictionary AuthenticationExtensions {
};

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

enum ScopedCredentialType {
    "ScopedCred"
};

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

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

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

typedef sequence < AAGUID > AuthenticatorSelectionList;

typedef BufferSource AAGUID;


   #promisesReferenced in:
     * 4.1.1. Create a new credential (makeCredential() method)
```



```
};

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

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

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

dictionary AuthenticationExtensions {
};

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

enum ScopedCredentialType {
    "ScopedCred"
};

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

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

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

typedef sequence < AAGUID > AuthenticatorSelectionList;

typedef BufferSource AAGUID;


   #promisesReferenced in:
     * 4.1.1. Create a new credential (makeCredential() method)
```

* 4.1.2. Use an existing credential (getAssertion() method)

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

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

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

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

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

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

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

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

#authenticatorReferenced in:
* 1. Introduction (2) (3) (4)
* 1.1. Use Cases
* 2. Conformance
* 3. Terminology (2) (3) (4) (5) (6) (7) (8) (9)
* 5. WebAuthn Authenticator model
* 5.2. Signature Format
* 5.2.1. Authenticator data
* 5.2.3. Generating a signature
* 5.3. Credential Attestation
* 5.3.4.1. Privacy
* 5.3.4.2. Attestation Certificate and Attestation Certificate CA Compromise
* 7.2. Packed Attestation Statement Format
* 7.4. Android Key Attestation Statement Format
* 7.5. Android SafetyNet Attestation Statement Format
* 9.4. SupportedExtensions Extension
* 9.5. User Verification Index (UVI) Extension

* 4.1.2. Use an existing credential (getAssertion() method)

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

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

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

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

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

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

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

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

#authenticatorReferenced in:
* 1. Introduction (2) (3) (4)
* 1.1. Use Cases
* 2. Conformance
* 3. Terminology (2) (3) (4) (5) (6) (7) (8) (9)
* 5. WebAuthn Authenticator model
* 5.2. Signature Format
* 5.2.1. Authenticator data
* 5.2.3. Generating a signature
* 5.3. Credential Attestation
* 5.3.4.1. Privacy
* 5.3.4.2. Attestation Certificate and Attestation Certificate CA Compromise
* 7.2. Packed Attestation Statement Format
* 7.4. Android Key Attestation Statement Format
* 7.5. Android SafetyNet Attestation Statement Format
* 9.4. SupportedExtensions Extension
* 9.5. User Verification Index (UVI) Extension

```
      * 9.6. Location Extension (2) (3) (4)
      * 9.7. User Verification Mode (UVM) Extension
      * 11. Sample scenarios

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

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

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

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

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

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

    #relying-partyReferenced in:
      * 1. Introduction (2) (3) (4)
      * 3. Terminology (2) (3) (4) (5) (6) (7) (8)
      * 4.2. Information about Scoped Credential (interface
        ScopedCredentialInfo)
      * 4.3. User Account Information (dictionary Account)
      * 4.5. Additional options for Credential Generation (dictionary
        ScopedCredentialOptions)
      * 5. WebAuthn Authenticator model
      * 5.2. Signature Format
      * 5.3. Credential Attestation
      * 6. Relying Party Operations
      * 8.3. Extending request parameters
      * 8.4. Extending client processing
      * 8.6. Example extension
      * 11.2. Authentication
      * 11.3. Decommissioning

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

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

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



```
      * 9.6. Location Extension (2) (3) (4)
      * 9.7. User Verification Mode (UVM) Extension
      * 11. Sample scenarios

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

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

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

    #credential-public-keyReferenced in:
      * 3. Terminology
      * 4.2. Information about Scoped Credential (interface
        ScopedCredentialInfo)
      * 5.2.1. Authenticator data

      * 7.4. Android Key Attestation Statement Format
      * 11.1. Registration (2)

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

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

    #relying-partyReferenced in:
      * 1. Introduction (2) (3) (4)
      * 3. Terminology (2) (3) (4) (5) (6) (7) (8)
      * 4.2. Information about Scoped Credential (interface
        ScopedCredentialInfo)
      * 4.3. User Account Information (dictionary Account)
      * 4.5. Additional options for Credential Generation (dictionary
        ScopedCredentialOptions)
      * 5. WebAuthn Authenticator model
      * 5.2. Signature Format
      * 5.3. Credential Attestation
      * 6. Relying Party Operations
      * 8.3. Extending request parameters
      * 8.4. Extending client processing
      * 8.6. Example extension
      * 11.2. Authentication
      * 11.3. Decommissioning

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

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

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

#user-consentReferenced in:
  * 1. Introduction

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

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

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

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

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

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

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

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

#dom-webauthentication-getassertion-assertionchallenge-options-assertio
nchallengeReferenced in:
  * 4.1.2. Use an existing credential (getAssertion() method)

#dom-webauthentication-getassertion-assertionchallenge-options-optionsR
eferenced in:
  * 4.1.2. Use an existing credential (getAssertion() method) (2) (3)
    (4) (5)

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

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

#user-consentReferenced in:
  * 1. Introduction

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

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

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

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

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

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

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

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

#dom-webauthentication-getassertion-assertionchallenge-options-assertio
nchallengeReferenced in:
  * 4.1.2. Use an existing credential (getAssertion() method)

#dom-webauthentication-getassertion-assertionchallenge-options-optionsR
eferenced in:
  * 4.1.2. Use an existing credential (getAssertion() method) (2) (3)
    (4) (5)

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

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

```
      * 4.1. WebAuthentication Interface
      * 4.1.1. Create a new credential (makeCredential() method) (2)
      * 4.8. Authentication Assertion Extensions (dictionary
        AuthenticationExtensions)
      * 4.9.4. Credential Descriptor (dictionary
        ScopedCredentialDescriptor)
      * 6. Relying Party Operations
      * 8. WebAuthn Extensions (2)
      * 8.2. Defining extensions
      * 8.3. Extending request parameters (2)
      * 9.1. FIDO AppId

    #scopedcredentialinfoReferenced in:
      * 4.1. WebAuthentication Interface
      * 4.1.1. Create a new credential (makeCredential() method) (2)
      * 4.2. Information about Scoped Credential (interface
        ScopedCredentialInfo)
      * 6. Relying Party Operations
      * 6.1. Registering a new credential (2) (3)

    #dom-scopedcredentialinfo-clientdataReferenced in:
      * 6.1. Registering a new credential (2)

    #dom-scopedcredentialinfo-attestationobjectReferenced in:
      * 6.1. Registering a new credential

    #dictdef-accountReferenced in:
      * 4.1. WebAuthentication Interface
      * 4.3. User Account Information (dictionary Account)
      * 5.1.1. The authenticatorMakeCredential operation

    #dom-account-idReferenced in:
      * 4.1.1. Create a new credential (makeCredential() method)
      * 4.3. User Account Information (dictionary Account)
      * 5.1.1. The authenticatorMakeCredential operation (2)

    #dictdef-scopedcredentialparametersReferenced in:
      * 4.1. WebAuthentication Interface
      * 4.1.1. Create a new credential (makeCredential() method)
      * 4.4. Parameters for Credential Generation (dictionary
        ScopedCredentialParameters)

    #dictdef-scopedcredentialoptionsReferenced in:
      * 4.1. WebAuthentication Interface
      * 4.5. Additional options for Credential Generation (dictionary
        ScopedCredentialOptions)

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

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

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

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

    #dom-scopedcredentialoptions-extensionsReferenced in:
      * 4.1.1. Create a new credential (makeCredential() method)
      * 8.3. Extending request parameters
```



```
      * 4.1. WebAuthentication Interface
      * 4.1.1. Create a new credential (makeCredential() method) (2)
      * 4.8. Authentication Assertion Extensions (dictionary
        AuthenticationExtensions)
      * 4.9.4. Credential Descriptor (dictionary
        ScopedCredentialDescriptor)
      * 6. Relying Party Operations
      * 8. WebAuthn Extensions (2)
      * 8.2. Defining extensions
      * 8.3. Extending request parameters (2)
      * 9.1. FIDO AppId

    #scopedcredentialinfoReferenced in:
      * 4.1. WebAuthentication Interface
      * 4.1.1. Create a new credential (makeCredential() method) (2)
      * 4.2. Information about Scoped Credential (interface
        ScopedCredentialInfo)
      * 6. Relying Party Operations
      * 6.1. Registering a new credential (2) (3)

    #dom-scopedcredentialinfo-clientdataReferenced in:
      * 6.1. Registering a new credential (2)

    #dom-scopedcredentialinfo-attestationobjectReferenced in:
      * 6.1. Registering a new credential

    #dictdef-accountReferenced in:
      * 4.1. WebAuthentication Interface
      * 4.3. User Account Information (dictionary Account)
      * 5.1.1. The authenticatorMakeCredential operation

    #dom-account-idReferenced in:
      * 4.1.1. Create a new credential (makeCredential() method)
      * 4.3. User Account Information (dictionary Account)
      * 5.1.1. The authenticatorMakeCredential operation (2)

    #dictdef-scopedcredentialparametersReferenced in:
      * 4.1. WebAuthentication Interface
      * 4.1.1. Create a new credential (makeCredential() method)
      * 4.4. Parameters for Credential Generation (dictionary
        ScopedCredentialParameters)

    #dictdef-scopedcredentialoptionsReferenced in:
      * 4.1. WebAuthentication Interface
      * 4.5. Additional options for Credential Generation (dictionary
        ScopedCredentialOptions)

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

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

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

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

    #dom-scopedcredentialoptions-extensionsReferenced in:
      * 4.1.1. Create a new credential (makeCredential() method)
      * 8.3. Extending request parameters
```

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

#attachment-platform-authenticatorsReferenced in:
  * 4.5.1. Credential Attachment enumeration (enum Attachment) (2)

#attachment-roaming-authenticatorsReferenced in:
  * 1.1.3. Other use cases and configurations
  * 4.5.1. Credential Attachment enumeration (enum Attachment) (2)

#attachment-platform-attachmentReferenced in:
  * 4.5.1. Credential Attachment enumeration (enum Attachment)

#attachment-cross-platform-attachedReferenced in:
  * 4.5.1. Credential Attachment enumeration (enum Attachment) (2)

#authenticationassertionReferenced in:
  * 3. Terminology
  * 4.1. WebAuthentication Interface
  * 4.1.2. Use an existing credential (getAssertion() method)
  * 4.6. Web Authentication Assertion (interface
    AuthenticationAssertion)
  * 6. Relying Party Operations
  * 6.2. Verifying an authentication assertion (2)

#dom-authenticationassertion-credentialReferenced in:
  * 6.2. Verifying an authentication assertion

#dom-authenticationassertion-clientdataReferenced in:
  * 6.2. Verifying an authentication assertion

#dom-authenticationassertion-authenticatordataReferenced in:
  * 6.2. Verifying an authentication assertion

#dom-authenticationassertion-signatureReferenced in:
  * 6.2. Verifying an authentication assertion

#dictdef-assertionoptionsReferenced in:
  * 4.1. WebAuthentication Interface
  * 4.7. Additional options for Assertion Generation (dictionary
    AssertionOptions)

#dom-assertionoptions-timeoutsecondsReferenced in:
  * 4.1.2. Use an existing credential (getAssertion() method) (2)

#dom-assertionoptions-rpidReferenced in:
  * 4.1.2. Use an existing credential (getAssertion() method) (2) (3)
  * 9.1. FIDO AppId

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

#dom-assertionoptions-extensionsReferenced in:
  * 4.1.2. Use an existing credential (getAssertion() method)

#dictdef-authenticationextensionsReferenced in:
  * 4.5. Additional options for Credential Generation (dictionary
    ScopedCredentialOptions)
```



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

#attachment-platform-authenticatorsReferenced in:
  * 4.5.1. Credential Attachment enumeration (enum Attachment) (2)

#attachment-roaming-authenticatorsReferenced in:
  * 1.1.3. Other use cases and configurations
  * 4.5.1. Credential Attachment enumeration (enum Attachment) (2)

#attachment-platform-attachmentReferenced in:
  * 4.5.1. Credential Attachment enumeration (enum Attachment)

#attachment-cross-platform-attachedReferenced in:
  * 4.5.1. Credential Attachment enumeration (enum Attachment) (2)

#authenticationassertionReferenced in:
  * 3. Terminology
  * 4.1. WebAuthentication Interface
  * 4.1.2. Use an existing credential (getAssertion() method)
  * 4.6. Web Authentication Assertion (interface
    AuthenticationAssertion)
  * 6. Relying Party Operations
  * 6.2. Verifying an authentication assertion (2)

#dom-authenticationassertion-credentialReferenced in:
  * 6.2. Verifying an authentication assertion

#dom-authenticationassertion-clientdataReferenced in:
  * 6.2. Verifying an authentication assertion

#dom-authenticationassertion-authenticatordataReferenced in:
  * 6.2. Verifying an authentication assertion

#dom-authenticationassertion-signatureReferenced in:
  * 6.2. Verifying an authentication assertion

#dictdef-assertionoptionsReferenced in:
  * 4.1. WebAuthentication Interface
  * 4.7. Additional options for Assertion Generation (dictionary
    AssertionOptions)

#dom-assertionoptions-timeoutsecondsReferenced in:
  * 4.1.2. Use an existing credential (getAssertion() method) (2)

#dom-assertionoptions-rpidReferenced in:
  * 4.1.2. Use an existing credential (getAssertion() method) (2) (3)
  * 9.1. FIDO AppId

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

#dom-assertionoptions-extensionsReferenced in:
  * 4.1.2. Use an existing credential (getAssertion() method)

#dictdef-authenticationextensionsReferenced in:
  * 4.5. Additional options for Credential Generation (dictionary
    ScopedCredentialOptions)
```

```
     * 4.7. Additional options for Assertion Generation (dictionary
       AssertionOptions)
     * 4.8. Authentication Assertion Extensions (dictionary
       AuthenticationExtensions)
     * 4.9.1. Client data used in WebAuthn signatures (dictionary
       ClientData)

   #dictdef-clientdataReferenced in:
     * 4.1.1. Create a new credential (makeCredential() method)
     * 4.1.2. Use an existing credential (getAssertion() method)
     * 4.9.1. Client data used in WebAuthn signatures (dictionary
       ClientData) (2)
     * 5.1.1. The authenticatorMakeCredential operation
     * 5.1.2. The authenticatorGetAssertion operation
     * 5.2. Signature Format
     * 5.2.1. Authenticator data
     * 5.2.4. Verifying a signature (2) (3) (4) (5) (6)
     * 6.1. Registering a new credential (2) (3) (4) (5) (6)
     * 7.2. Packed Attestation Statement Format
     * 8. WebAuthn Extensions (2)
     * 8.2. Defining extensions (2)
     * 8.4. Extending client processing (2)

   #dom-clientdata-challengeReferenced in:
     * 5.2.4. Verifying a signature
     * 6.1. Registering a new credential

   #dom-clientdata-originReferenced in:
     * 5.2.4. Verifying a signature
     * 6.1. Registering a new credential

   #dom-clientdata-hashalgReferenced in:
     * 4.1.1. Create a new credential (makeCredential() method)
     * 4.1.2. Use an existing credential (getAssertion() method)
     * 5.2.4. Verifying a signature
     * 6.1. Registering a new credential

   #dom-clientdata-tokenbindingReferenced in:
     * 5.2.4. Verifying a signature
     * 6.1. Registering a new credential

   #dom-clientdata-extensionsReferenced in:
     * 5.2.4. Verifying a signature
     * 6.1. Registering a new credential
     * 8.4. Extending client processing

   #clientdata-hashalgReferenced in:
     * 4.9.1. Client data used in WebAuthn signatures (dictionary
       ClientData)

   #clientdatajsonReferenced in:
     * 4.1.1. Create a new credential (makeCredential() method) (2)
     * 4.1.2. Use an existing credential (getAssertion() method) (2)
     * 4.2. Information about Scoped Credential (interface
       ScopedCredentialInfo)
     * 4.9.1. Client data used in WebAuthn signatures (dictionary
       ClientData)
     * 5.2.4. Verifying a signature

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



```
     * 4.7. Additional options for Assertion Generation (dictionary
       AssertionOptions)
     * 4.8. Authentication Assertion Extensions (dictionary
       AuthenticationExtensions)
     * 4.9.1. Client data used in WebAuthn signatures (dictionary
       ClientData)

   #dictdef-clientdataReferenced in:
     * 4.1.1. Create a new credential (makeCredential() method)
     * 4.1.2. Use an existing credential (getAssertion() method)
     * 4.9.1. Client data used in WebAuthn signatures (dictionary
       ClientData) (2)
     * 5.1.1. The authenticatorMakeCredential operation
     * 5.1.2. The authenticatorGetAssertion operation
     * 5.2. Signature Format
     * 5.2.1. Authenticator data
     * 5.2.4. Verifying a signature (2) (3) (4) (5) (6)
     * 6.1. Registering a new credential (2) (3) (4) (5) (6)

     * 8. WebAuthn Extensions (2)
     * 8.2. Defining extensions (2)
     * 8.4. Extending client processing (2)

   #dom-clientdata-challengeReferenced in:
     * 5.2.4. Verifying a signature
     * 6.1. Registering a new credential

   #dom-clientdata-originReferenced in:
     * 5.2.4. Verifying a signature
     * 6.1. Registering a new credential

   #dom-clientdata-hashalgReferenced in:
     * 4.1.1. Create a new credential (makeCredential() method)
     * 4.1.2. Use an existing credential (getAssertion() method)
     * 5.2.4. Verifying a signature
     * 6.1. Registering a new credential

   #dom-clientdata-tokenbindingReferenced in:
     * 5.2.4. Verifying a signature
     * 6.1. Registering a new credential

   #dom-clientdata-extensionsReferenced in:
     * 5.2.4. Verifying a signature
     * 6.1. Registering a new credential
     * 8.4. Extending client processing

   #clientdata-hashalgReferenced in:
     * 4.9.1. Client data used in WebAuthn signatures (dictionary
       ClientData)

   #clientdatajsonReferenced in:
     * 4.1.1. Create a new credential (makeCredential() method) (2)
     * 4.1.2. Use an existing credential (getAssertion() method) (2)
     * 4.2. Information about Scoped Credential (interface
       ScopedCredentialInfo)
     * 4.9.1. Client data used in WebAuthn signatures (dictionary
       ClientData)
     * 5.2.4. Verifying a signature

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

* 4.2. Information about Scoped Credential (interface ScopedCredentialInfo)
* 4.9.1. Client data used in WebAuthn signatures (dictionary ClientData)
* 5.1.1. The authenticatorMakeCredential operation (2)
* 5.1.2. The authenticatorGetAssertion operation (2)
* 5.2. Signature Format
* 5.2.3. Generating a signature (2) (3)
* 5.2.4. Verifying a signature (2)
* 5.3.1. Attestation Statement Formats (2)
* 5.3.3. Generating an Attestation Object
* 6.1. Registering a new credential (2)
* 7.2. Packed Attestation Statement Format (2) (3) (4) (5) (6)
* 7.3. TPM Attestation Statement Format (2)
* 7.4. Android Key Attestation Statement Format (2)
* 7.5. Android SafetyNet Attestation Statement Format (2)
* 7.6. FIDO U2F Attestation Statement Format (2)

#enumdef-scopedcredentialtypeReferenced in:
* 4.1.1. Create a new credential (makeCredential() method)
* 4.4. Parameters for Credential Generation (dictionary ScopedCredentialParameters)
* 4.9.2. Credential Type enumeration (enum ScopedCredentialType)
* 4.9.3. Unique Identifier for Credential (interface ScopedCredential) (2)
* 4.9.4. Credential Descriptor (dictionary ScopedCredentialDescriptor)
* 5.1.1. The authenticatorMakeCredential operation (2) (3)

#scopedcredentialReferenced in:
* 4.6. Web Authentication Assertion (interface AuthenticationAssertion)
* 4.9.3. Unique Identifier for Credential (interface ScopedCredential)
* 4.9.4. Credential Descriptor (dictionary ScopedCredentialDescriptor)
* 5.1.1. The authenticatorMakeCredential operation (2)

#dictdef-scopedcredentialdescriptorReferenced in:
* 4.5. Additional options for Credential Generation (dictionary ScopedCredentialOptions)
* 4.7. Additional options for Assertion Generation (dictionary AssertionOptions)
* 4.9.4. Credential Descriptor (dictionary ScopedCredentialDescriptor)

#enumdef-transportReferenced in:
* 4.9.4. Credential Descriptor (dictionary ScopedCredentialDescriptor)

#authenticatormakecredentialReferenced in:
* 3. Terminology (2)
* 4.1.1. Create a new credential (makeCredential() method)
* 5.1.3. The authenticatorCancel operation (2)
* 8. WebAuthn Extensions
* 8.2. Defining extensions

#authenticatorgetassertionReferenced in:
* 3. Terminology (2)
* 4.1.2. Use an existing credential (getAssertion() method)
* 5.1.3. The authenticatorCancel operation (2)
* 5.2.1. Authenticator data

* 4.2. Information about Scoped Credential (interface ScopedCredentialInfo)
* 4.9.1. Client data used in WebAuthn signatures (dictionary ClientData)
* 5.1.1. The authenticatorMakeCredential operation (2)
* 5.1.2. The authenticatorGetAssertion operation (2)
* 5.2. Signature Format
* 5.2.3. Generating a signature (2) (3)
* 5.2.4. Verifying a signature (2)
* 5.3.1. Attestation Statement Formats (2)
* 5.3.3. Generating an Attestation Object
* 6.1. Registering a new credential (2)
* 7.2. Packed Attestation Statement Format (2) (3) (4) (5) (6)
* 7.3. TPM Attestation Statement Format (2)
* 7.4. Android Key Attestation Statement Format (2)
* 7.5. Android SafetyNet Attestation Statement Format (2)
* 7.6. FIDO U2F Attestation Statement Format (2)

#enumdef-scopedcredentialtypeReferenced in:
* 4.1.1. Create a new credential (makeCredential() method)
* 4.4. Parameters for Credential Generation (dictionary ScopedCredentialParameters)
* 4.9.2. Credential Type enumeration (enum ScopedCredentialType)
* 4.9.3. Unique Identifier for Credential (interface ScopedCredential) (2)
* 4.9.4. Credential Descriptor (dictionary ScopedCredentialDescriptor)
* 5.1.1. The authenticatorMakeCredential operation (2) (3)

#scopedcredentialReferenced in:
* 4.6. Web Authentication Assertion (interface AuthenticationAssertion)
* 4.9.3. Unique Identifier for Credential (interface ScopedCredential)
* 4.9.4. Credential Descriptor (dictionary ScopedCredentialDescriptor)
* 5.1.1. The authenticatorMakeCredential operation (2)

#dictdef-scopedcredentialdescriptorReferenced in:
* 4.5. Additional options for Credential Generation (dictionary ScopedCredentialOptions)
* 4.7. Additional options for Assertion Generation (dictionary AssertionOptions)
* 4.9.4. Credential Descriptor (dictionary ScopedCredentialDescriptor)

#enumdef-transportReferenced in:
* 4.9.4. Credential Descriptor (dictionary ScopedCredentialDescriptor)

#authenticatormakecredentialReferenced in:
* 3. Terminology (2)
* 4.1.1. Create a new credential (makeCredential() method)
* 5.1.3. The authenticatorCancel operation (2)
* 8. WebAuthn Extensions
* 8.2. Defining extensions

#authenticatorgetassertionReferenced in:
* 3. Terminology (2)
* 4.1.2. Use an existing credential (getAssertion() method)
* 5.1.3. The authenticatorCancel operation (2)
* 5.2.1. Authenticator data

```
      * 5.2.3. Generating a signature
      * 8. WebAuthn Extensions
      * 8.2. Defining extensions

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

    #authenticatordataReferenced in:
      * 5.1.1. The authenticatorMakeCredential operation (2)
      * 5.1.2. The authenticatorGetAssertion operation (2) (3)
      * 5.2.1. Authenticator data (2)
      * 5.2.2. Attestation data
      * 5.2.3. Generating a signature (2) (3)
      * 5.2.4. Verifying a signature
      * 5.3.1. Attestation Statement Formats (2)
      * 5.3.3. Generating an Attestation Object (2)
      * 5.3.4.3. Attestation Certificate Hierarchy
      * 7.2. Packed Attestation Statement Format (2) (3) (4) (5) (6) (7)
        (8) (9)
      * 7.3. TPM Attestation Statement Format (2) (3) (4)
      * 7.4. Android Key Attestation Statement Format (2) (3) (4) (5)
      * 7.5. Android SafetyNet Attestation Statement Format (2)
      * 7.6. FIDO U2F Attestation Statement Format (2) (3)
      * 8. WebAuthn Extensions (2)
      * 8.2. Defining extensions (2)
      * 8.5. Extending authenticator processing (2)
      * 8.6. Example extension
      * 9.5. User Verification Index (UVI) Extension
      * 9.6. Location Extension
      * 9.7. User Verification Mode (UVM) Extension

    #attestation-statement-formatReferenced in:
      * 3. Terminology
      * 4.2. Information about Scoped Credential (interface
        ScopedCredentialInfo)
      * 5.3.3. Generating an Attestation Object (2)

    #attestation-typeReferenced in:
      * 3. Terminology

    #basic-attestationReferenced in:
      * 5.3.4.1. Privacy

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

    #privacy-caReferenced in:
      * 5.3.4.1. Privacy

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

    #atttobesignedReferenced in:
      * 5.3.3. Generating an Attestation Object

    #attestation-format-identifierReferenced in:
      * 5.3.1. Attestation Statement Formats
      * 5.3.3. Generating an Attestation Object
```

/Users/jehodges/Documents/work/standards/W3C/webauthn/index-vgb-u2f-attestation-dc90eab.txt, Top line: 4320

```
      * 5.2.3. Generating a signature
      * 8. WebAuthn Extensions
      * 8.2. Defining extensions

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

    #authenticatordataReferenced in:
      * 5.1.1. The authenticatorMakeCredential operation (2)
      * 5.1.2. The authenticatorGetAssertion operation (2) (3)
      * 5.2.1. Authenticator data (2)
      * 5.2.2. Attestation data
      * 5.2.3. Generating a signature (2) (3)
      * 5.2.4. Verifying a signature
      * 5.3.1. Attestation Statement Formats (2)
      * 5.3.3. Generating an Attestation Object (2)
      * 5.3.4.3. Attestation Certificate Hierarchy
      * 7.2. Packed Attestation Statement Format (2) (3) (4) (5) (6) (7)
        (8)
      * 7.3. TPM Attestation Statement Format (2) (3) (4)
      * 7.4. Android Key Attestation Statement Format (2) (3) (4) (5)
      * 7.5. Android SafetyNet Attestation Statement Format (2)
      * 7.6. FIDO U2F Attestation Statement Format (2) (3)
      * 8. WebAuthn Extensions (2)
      * 8.2. Defining extensions (2)
      * 8.5. Extending authenticator processing (2)
      * 8.6. Example extension
      * 9.5. User Verification Index (UVI) Extension
      * 9.6. Location Extension
      * 9.7. User Verification Mode (UVM) Extension

    #attestation-statement-formatReferenced in:
      * 3. Terminology
      * 4.2. Information about Scoped Credential (interface
        ScopedCredentialInfo)
      * 5.3.3. Generating an Attestation Object (2)

    #attestation-typeReferenced in:
      * 3. Terminology

    #basic-attestationReferenced in:
      * 5.3.4.1. Privacy

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

    #privacy-caReferenced in:
      * 5.3.4.1. Privacy

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


    #attestation-format-identifierReferenced in:
      * 5.3.1. Attestation Statement Formats
      * 5.3.3. Generating an Attestation Object
```

#client-argumentReferenced in:
  * 8.3. Extending request parameters

#contentReferenced in:
  * 9.2. Transaction authorization (2) (3) (4) (5)

#typedefdef-aaguidReferenced in:
  * 9.3. Authenticator Selection Extension

#client-argumentReferenced in:
  * 8.3. Extending request parameters

#contentReferenced in:
  * 9.2. Transaction authorization (2) (3) (4) (5)

#typedefdef-aaguidReferenced in:
  * 9.3. Authenticator Selection Extension