

0001 THE URL:file://localhost/Users/jehodges/documents/work/standards/W3C/webauthn/index-master-tr-dda3e24-WD-05.html

0002 THE TITLE:Web Authentication: An API for accessing Public Key Credentials

0003 ^I Jump to Table of Contents-> Pop Out Sidebar

0004

0005 W3C

0006

0007 Web Authentication: An API for accessing Public Key Credentials

0008

0009 W3C Working Draft, 5 May 2017

0010

0011 This version:

0012 <https://www.w3.org/TR/2017/WD-webauthn-20170505/>

0013

0014 Latest published version:

0015 <https://www.w3.org/TR/webauthn/>

0016

0017 Editor's Draft:

0018 <https://w3c.github.io/webauthn/>

0019

0020 Previous Versions:

0021 <https://www.w3.org/TR/2017/WD-webauthn-20170216/>

0022 <https://www.w3.org/TR/2016/WD-webauthn-20161207/>

0023 <https://www.w3.org/TR/2016/WD-webauthn-20160928/>

0024 <https://www.w3.org/TR/2016/WD-webauthn-20160902/>

0025 <https://www.w3.org/TR/2016/WD-webauthn-20160531/>

0026

0027 Issue Tracking:

0028 Github

0029

0030 Editors:

0031 Vijay Bharadwaj (Microsoft)

0032 Hubert Le Van Gong (PayPal)

0033 Dirk Balfanz (Google)

0034 Alexei Czeskis (Google)

0035 Arnar Birgisson (Google)

0036 Jeff Hodges (PayPal)

0037 Michael B. Jones (Microsoft)

0038 Rolf Lindemann (Nok Nok Labs)

0039 J.C. Jones (Mozilla)

0040

0041 Tests:

0042 web-platform-tests webauthn/ (ongoing work)

0043

0044 Copyright 2017 W3C^ (MIT, ERCIM, Keio, Beihang). W3C liability,

0045 trademark and document use rules apply.

0046

0047

0048 Abstract

0049

0050 This specification defines an API enabling the creation and use of

0051 strong, attested, scoped, public key-based credentials by web

0052 applications, for the purpose of strongly authenticating users.

0053 Conceptually, one or more credentials, each scoped to a given Relying

0054 Party, are created and stored on an authenticator by the user agent in

0055 conjunction with the web application. The user agent mediates access to

0056 public key credentials in order to preserve user privacy.

0057 Authenticators are responsible for ensuring that no operation is

0058 performed without user consent. Authenticators provide cryptographic

0059 proof of their properties to relying parties via attestation. This

0060 specification also describes the functional model for WebAuthn

0061 conformant authenticators, including their signature and attestation

0062 functionality.

0063

0064 Status of this document

0065

0066 This section describes the status of this document at the time of its

0067 publication. Other documents may supersede this document. A list of

0068 current W3C publications and the latest revision of this technical

0069

0001 THE URL:file://localhost/Users/jehodges/documents/work/standards/W3C/WebAuthn/index-master-tr-598ac41-WD-06.html

0002 THE TITLE:Web Authentication: An API for accessing Public Key Credentials Level 1

0003 ^I Jump to Table of Contents-> Pop Out Sidebar

0004

0005 W3C

0006

0007 Web Authentication: An API for accessing Public Key Credentials Level 1

0008

0009 W3C Working Draft, 11 August 2017

0010

0011 This version:

0012 <https://www.w3.org/TR/2017/WD-webauthn-20170811/>

0013

0014 Latest published version:

0015 <https://www.w3.org/TR/webauthn/>

0016

0017 Editor's Draft:

0018 <https://w3c.github.io/webauthn/>

0019

0020 Previous Versions:

0021 <https://www.w3.org/TR/2017/WD-webauthn-20170505/>

0022 <https://www.w3.org/TR/2017/WD-webauthn-20170216/>

0023 <https://www.w3.org/TR/2016/WD-webauthn-20161207/>

0024 <https://www.w3.org/TR/2016/WD-webauthn-20160928/>

0025 <https://www.w3.org/TR/2016/WD-webauthn-20160902/>

0026 <https://www.w3.org/TR/2016/WD-webauthn-20160531/>

0027

0028 Issue Tracking:

0029 Github

0030

0031 Editors:

0032 Vijay Bharadwaj (Microsoft)

0033 Hubert Le Van Gong (PayPal)

0034 Dirk Balfanz (Google)

0035 Alexei Czeskis (Google)

0036 Arnar Birgisson (Google)

0037 Jeff Hodges (PayPal)

0038 Michael B. Jones (Microsoft)

0039 Rolf Lindemann (Nok Nok Labs)

0040 J.C. Jones (Mozilla)

0041

0042 Tests:

0043 web-platform-tests webauthn/ (ongoing work)

0044

0045 Copyright 2017 W3C^ (MIT, ERCIM, Keio, Beihang). W3C liability,

0046 trademark and document use rules apply.

0047

0048

0049 Abstract

0050

0051 This specification defines an API enabling the creation and use of

0052 strong, attested, scoped, public key-based credentials by web

0053 applications, for the purpose of strongly authenticating users.

0054 Conceptually, one or more public key credentials, each scoped to a

0055 given Relying Party, are created and stored on an authenticator by the

0056 user agent in conjunction with the web application. The user agent

0057 mediates access to public key credentials in order to preserve user

0058 privacy. Authenticators are responsible for ensuring that no operation

0059 is performed without user consent. Authenticators provide cryptographic

0060 proof of their properties to relying parties via attestation. This

0061 specification also describes the functional model for WebAuthn

0062 conformant authenticators, including their signature and attestation

0063 functionality.

0064

0065 Status of this document

0066

0067 This section describes the status of this document at the time of its

0068 publication. Other documents may supersede this document. A list of

0069 current W3C publications and the latest revision of this technical

0066 report can be found in the W3C technical reports index at
0070 https://www.w3.org/TR/.
0071
0072 This document was published by the Web Authentication Working Group as
0073 a Working Draft. This document is intended to become a W3C
0074 Recommendation. Feedback and comments on this specification are
0075 welcome. Please use Github issues. Discussions may also be found in the
0076 public-webauthn@w3.org archives.
0077
0078 Publication as a Working Draft does not imply endorsement by the W3C
0079 Membership. This is a draft document and may be updated, replaced or
0080 obsoleted by other documents at any time. It is inappropriate to cite
0081 this document as other than work in progress.
0082
0083 This document was produced by a group operating under the 5 February
0084 2004 W3C Patent Policy. W3C maintains a public list of any patent
0085 disclosures made in connection with the deliverables of the group; that
0086 page also includes instructions for disclosing a patent. An individual
0087 who has actual knowledge of a patent which the individual believes
0088 contains Essential Claim(s) must disclose the information in accordance
0089 with section 6 of the W3C Patent Policy.
0090
0091 This document is governed by the 1 March 2017 W3C Process Document.
0092
0093 Table of Contents
0094
0095 1. 1 Introduction
0096 1. 1.1 Use Cases
0097 1. 1.1.1 Registration
0098 2. 1.1.2 Authentication
0099 3. 1.1.3 Other use cases and configurations
0100 2. 2 Conformance
0101 1. 2.1 Dependencies
0102 3. 3 Terminology
0103 4. 4 Web Authentication API
0104 1. 4.1 PublicKeyCredential Interface
0105 1. 4.1.1 CredentialRequestOptions Extension
0106 2. 4.1.2 CredentialCreationOptions Extension
0107 3. 4.1.3 Create a new credential - PublicKeyCredential's
0108 [[Create]](options) method
0109 4. 4.1.4 Use an existing credential -
0110 PublicKeyCredential::[[DiscoverFromExternalSource]](optio
0111 ns) method

0112 2. 4.2 Authenticator Responses (interface AuthenticatorResponse)
0113 1. 4.2.1 Information about Public Key Credential (interface
0114 AuthenticatorAttestationResponse)
0115 2. 4.2.2 Web Authentication Assertion (interface
0116 AuthenticatorAssertionResponse)
0117 3. 4.3 Parameters for Credential Generation (dictionary
0118 PublicKeyCredentialParameters)
0119 4. 4.4 User Account Parameters for Credential Generation

(dictionary PublicKeyCredentialUserEntity)
0120 5. 4.5 Options for Credential Creation (dictionary
0121 MakeCredentialOptions)
0122 1. 4.5.1 Entity Description
0123 2. 4.5.2 Authenticator Selection Criteria
0124 3. 4.5.3 Credential Attachment enumeration (enum Attachment)
0125 6. 4.6 Options for Assertion Generation (dictionary
0126 PublicKeyCredentialRequestOptions)
0127 7. 4.7 Authentication Extensions (typedef
0128 AuthenticationExtensions)
0129 8. 4.8 Supporting Data Structures
0130 1. 4.8.1 Client data used in WebAuthn signatures (dictionary
0131

0070 report can be found in the W3C technical reports index at
0071 https://www.w3.org/TR/.
0072
0073 This document was published by the Web Authentication Working Group as
0074 a Working Draft. This document is intended to become a W3C
0075 Recommendation. Feedback and comments on this specification are
0076 welcome. Please use Github issues. Discussions may also be found in the
0077 public-webauthn@w3.org archives.
0078
0079 Publication as a Working Draft does not imply endorsement by the W3C
0080 Membership. This is a draft document and may be updated, replaced or
0081 obsoleted by other documents at any time. It is inappropriate to cite
0082 this document as other than work in progress.
0083
0084 This document was produced by a group operating under the 5 February
0085 2004 W3C Patent Policy. W3C maintains a public list of any patent
0086 disclosures made in connection with the deliverables of the group; that
0087 page also includes instructions for disclosing a patent. An individual
0088 who has actual knowledge of a patent which the individual believes
0089 contains Essential Claim(s) must disclose the information in accordance
0090 with section 6 of the W3C Patent Policy.
0091
0092 This document is governed by the 1 March 2017 W3C Process Document.
0093
0094 Table of Contents
0095
0096 1. 1 Introduction
0097 1. 1.1 Use Cases
0098 1. 1.1.1 Registration
0099 2. 1.1.2 Authentication
0100 3. 1.1.3 Other use cases and configurations
0101 2. 2 Conformance
0102 1. 2.1 Dependencies
0103 3. 3 Terminology
0104 4. 4 Web Authentication API
0105 1. 4.1 PublicKeyCredential Interface
0106 1. 4.1.1 CredentialCreationOptions Extension
0107 2. 4.1.2 CredentialRequestOptions Extension
0108 3. 4.1.3 Create a new credential - PublicKeyCredential's
0109 [[Create]](options) method
0110 4. 4.1.4 Use an existing credential to make an assertion -
0111 PublicKeyCredential's
0112 [[DiscoverFromExternalSource]](options) method
0113 5. 4.1.5 Platform Authenticator Availability -
0114 PublicKeyCredential's isPlatformAuthenticatorAvailable()
0115 method
0116 2. 4.2 Authenticator Responses (interface AuthenticatorResponse)
0117 1. 4.2.1 Information about Public Key Credential (interface
0118 AuthenticatorAttestationResponse)
0119 2. 4.2.2 Web Authentication Assertion (interface
0120 AuthenticatorAssertionResponse)
0121 3. 4.3 Parameters for Credential Generation (dictionary
0122 PublicKeyCredentialParameters)
0123 4. 4.4 Options for Credential Creation (dictionary
0124 MakePublicKeyCredentialOptions)
0125 1. 4.4.1 Public Key Entity Description (dictionary
0126 PublicKeyCredentialEntity)
0127 2. 4.4.2 User Account Parameters for Credential Generation
0128 (dictionary PublicKeyCredentialUserEntity)
0129 3. 4.4.3 Authenticator Selection Criteria (dictionary
0130 AuthenticatorSelectionCriteria)
0131 4. 4.4.4 Authenticator Attachment enumeration (enum
0132 AuthenticatorAttachment)
0133 5. 4.5 Options for Assertion Generation (dictionary
0134 PublicKeyCredentialRequestOptions)
0135 6. 4.6 Authentication Extensions (typedef
0136 AuthenticationExtensions)
0137 7. 4.7 Supporting Data Structures
0138 1. 4.7.1 Client data used in WebAuthn signatures (dictionary

0132	CollectedClientData)
0133	2. 4.8.2 Credential Type enumeration (enum
0134	PublicKeyCredentialType)
0135	3. 4.8.3 Credential Descriptor (dictionary
0136	PublicKeyCredentialDescriptor)
0137	4. 4.8.4 Credential Transport enumeration (enum
0138	ExternalTransport)
0139	5. 4.8.5 Cryptographic Algorithm Identifier (type
0140	AlgorithmIdentifier)
0141	5. 5 WebAuthn Authenticator model
0142	1. 5.1 Authenticator data
0143	2. 5.2 Authenticator operations
0144	1. 5.2.1 The authenticatorMakeCredential operation
0145	2. 5.2.2 The authenticatorGetAssertion operation
0146	3. 5.2.3 The authenticatorCancel operation
0147	3. 5.3 Credential Attestation
0148	1. 5.3.1 Attestation data
0149	2. 5.3.2 Attestation Statement Formats
0150	3. 5.3.3 Attestation Types
0151	4. 5.3.4 Generating an Attestation Object
0152	5. 5.3.5 Security Considerations
0153	1. 5.3.5.1 Privacy
0154	2. 5.3.5.2 Attestation Certificate and Attestation
0155	Certificate CA Compromise
0156	3. 5.3.5.3 Attestation Certificate Hierarchy
0157	6. 6 Relying Party Operations
0158	1. 6.1 Registering a new credential
0159	2. 6.2 Verifying an authentication assertion
0160	7. 7 Defined Attestation Statement Formats
0161	1. 7.1 Attestation Statement Format Identifiers
0162	2. 7.2 Packed Attestation Statement Format
0163	1. 7.2.1 Packed attestation statement certificate
0164	requirements
0165	3. 7.3 TPM Attestation Statement Format
0166	1. 7.3.1 TPM attestation statement certificate requirements
0167	4. 7.4 Android Key Attestation Statement Format
0168	5. 7.5 Android SafetyNet Attestation Statement Format
0169	6. 7.6 FIDO U2F Attestation Statement Format
0170	8. 8 WebAuthn Extensions
0171	1. 8.1 Extension Identifiers
0172	2. 8.2 Defining extensions
0173	3. 8.3 Extending request parameters
0174	4. 8.4 Client extension processing
0175	5. 8.5 Authenticator extension processing
0176	6. 8.6 Example Extension
0177	9. 9 Defined Extensions
0178	1. 9.1 FIDO AppId Extension (appid)
0179	2. 9.2 Simple Transaction Authorization Extension (txAuthSimple)
0180	3. 9.3 Generic Transaction Authorization Extension
0181	(txAuthGeneric)
0182	4. 9.4 Authenticator Selection Extension (authnSel)
0183	5. 9.5 Supported Extensions Extension (exts)
0184	6. 9.6 User Verification Index Extension (uvi)
0185	7. 9.7 Location Extension (loc)
0186	8. 9.8 User Verification Method Extension (uvm)
0187	10. 10 IANA Considerations
0188	1. 10.1 WebAuthn Attestation Statement Format Identifier
0189	Registrations
0190	2. 10.2 WebAuthn Extension Identifier Registrations
0191	11. 11 Sample scenarios
0192	1. 11.1 Registration
0193	2. 11.2 Authentication
0194	3. 11.3 Decommissioning
0195	12. 12 Acknowledgements
0196	13. Index
0197	1. Terms defined by this specification
0198	2. Terms defined by reference
0199	14. References

0139	CollectedClientData)
0140	2. 4.7.2 Credential Type enumeration (enum
0141	PublicKeyCredentialType)
0142	3. 4.7.3 Credential Descriptor (dictionary
0143	PublicKeyCredentialDescriptor)
0144	4. 4.7.4 Authenticator Transport enumeration (enum
0145	AuthenticatorTransport)
0146	5. 4.7.5 Cryptographic Algorithm Identifier (typedef
0147	COSEAlgorithmIdentifier)
0148	5. 5 WebAuthn Authenticator model
0149	1. 5.1 Authenticator data
0150	2. 5.2 Authenticator operations
0151	1. 5.2.1 The authenticatorMakeCredential operation
0152	2. 5.2.2 The authenticatorGetAssertion operation
0153	3. 5.2.3 The authenticatorCancel operation
0154	3. 5.3 Attestation
0155	1. 5.3.1 Attestation data
0156	2. 5.3.2 Attestation Statement Formats
0157	3. 5.3.3 Attestation Types
0158	4. 5.3.4 Generating an Attestation Object
0159	5. 5.3.5 Security Considerations
0160	1. 5.3.5.1 Privacy
0161	2. 5.3.5.2 Attestation Certificate and Attestation
0162	Certificate CA Compromise
0163	3. 5.3.5.3 Attestation Certificate Hierarchy
0164	6. 6 Relying Party Operations
0165	1. 6.1 Registering a new credential
0166	2. 6.2 Verifying an authentication assertion
0167	7. 7 Defined Attestation Statement Formats
0168	1. 7.1 Attestation Statement Format Identifiers
0169	2. 7.2 Packed Attestation Statement Format
0170	1. 7.2.1 Packed attestation statement certificate
0171	requirements
0172	3. 7.3 TPM Attestation Statement Format
0173	1. 7.3.1 TPM attestation statement certificate requirements
0174	4. 7.4 Android Key Attestation Statement Format
0175	5. 7.5 Android SafetyNet Attestation Statement Format
0176	6. 7.6 FIDO U2F Attestation Statement Format
0177	8. 8 WebAuthn Extensions
0178	1. 8.1 Extension Identifiers
0179	2. 8.2 Defining extensions
0180	3. 8.3 Extending request parameters
0181	4. 8.4 Client extension processing
0182	5. 8.5 Authenticator extension processing
0183	6. 8.6 Example Extension
0184	9. 9 Defined Extensions
0185	1. 9.1 FIDO AppId Extension (appid)
0186	2. 9.2 Simple Transaction Authorization Extension (txAuthSimple)
0187	3. 9.3 Generic Transaction Authorization Extension
0188	(txAuthGeneric)
0189	4. 9.4 Authenticator Selection Extension (authnSel)
0190	5. 9.5 Supported Extensions Extension (exts)
0191	6. 9.6 User Verification Index Extension (uvi)
0192	7. 9.7 Location Extension (loc)
0193	8. 9.8 User Verification Method Extension (uvm)
0194	10. 10 IANA Considerations
0195	1. 10.1 WebAuthn Attestation Statement Format Identifier
0196	Registrations
0197	2. 10.2 WebAuthn Extension Identifier Registrations
0198	3. 10.3 COSE Algorithm Registrations
0199	11. 11 Sample scenarios
0200	1. 11.1 Registration
0201	2. 11.2 Registration Specifically with Platform Authenticator
0202	3. 11.3 Authentication
0203	4. 11.4 Decommissioning
0204	12. 12 Acknowledgements
0205	13. Index
0206	1. Terms defined by this specification
0207	2. Terms defined by reference
0208	14. References

0200	1. Normative References
0201	2. Informative References
0202	15. IDL Index
0203	
0204	1. Introduction
0205	
0206	This section is not normative.
0207	
0208	This specification defines an API enabling the creation and use of
0209	strong, attested, scoped, public key-based credentials by web
0210	applications, for the purpose of strongly authenticating users. A
0211	public key credential is created and stored by an authenticator at the
0212	behest of a Relying Party, subject to user consent. Subsequently, the
0213	public key credential can only be accessed by origins belonging to that
0214	Relying Party. This scoping is enforced jointly by conforming User
0215	Agents and authenticators. Additionally, privacy across Relying Parties
0216	is maintained; Relying Parties are not able to detect any properties,
0217	or even the existence, of credentials scoped to other Relying Parties.
0218	
0219	Relying Parties employ the Web Authentication API during two distinct,
0220	but related, ceremonies involving a user. The first is Registration,
0221	where a public key credential is created on an authenticator, and
0222	associated by a Relying Party with the present user's account (the
0223	account may already exist or may be created at this time). The second
0224	is Authentication, where the Relying Party is presented with an
0225	Authentication Assertion proving the presence and consent of the user
0226	who registered the public key credential. Functionally, the Web
0227	Authentication API comprises a PublicKeyCredential which extends the
0228	Credential Management API [CREDENTIAL-MANAGEMENT-1], and infrastructure
0229	which allows those credentials to be used with
0230	navigator.credentials.create() and navigator.credentials.get(). The
0231	former is used during Registration, and the latter during
0232	Authentication.
0233	
0234	Broadly, compliant authenticators protect public key credentials, and
0235	interact with user agents to implement the Web Authentication API. Some
0236	authenticators may run on the same computing device (e.g., smart phone,
0237	tablet, desktop PC) as the user agent is running on. For instance, such
0238	an authenticator might consist of a Trusted Execution Environment (TEE)
0239	applet, a Trusted Platform Module (TPM), or a Secure Element (SE)
0240	integrated into the computing device in conjunction with some means for
0241	user verification, along with appropriate platform software to mediate
0242	access to these components' functionality. Other authenticators may
0243	operate autonomously from the computing device running the user agent,
0244	and be accessed over a transport such as Universal Serial Bus (USB),
0245	Bluetooth Low Energy (BLE) or Near Field Communications (NFC).
0246	
0247	1.1. Use Cases
0248	
0249	The below use case scenarios illustrate use of two very different types
0250	of authenticators, as well as outline further scenarios. Additional
0251	scenarios, including sample code, are given later in 11 Sample
0252	scenarios.
0253	
0254	1.1.1. Registration
0255	
0256	* On a phone:
0257	+ User navigates to example.com in a browser and signs in to an
0258	existing account using whatever method they have been using
0259	(possibly a legacy method such as a password), or creates a
0260	new account.
0261	+ The phone prompts, "Do you want to register this device with
0262	example.com?"
0263	+ User agrees.
0264	+ The phone prompts the user for a previously configured
0265	authorization gesture (PIN, biometric, etc.); the user
0266	provides this.
0267	+ Website shows message, "Registration complete."
0268	
0269	1.1.2. Authentication

0209	1. Normative References
0210	2. Informative References
0211	15. IDL Index
0212	
0213	1. Introduction
0214	
0215	This section is not normative.
0216	
0217	This specification defines an API enabling the creation and use of
0218	strong, attested, scoped, public key-based credentials by web
0219	applications, for the purpose of strongly authenticating users. A
0220	public key credential is created and stored by an authenticator at the
0221	behest of a Relying Party, subject to user consent. Subsequently, the
0222	public key credential can only be accessed by origins belonging to that
0223	Relying Party. This scoping is enforced jointly by conforming User
0224	Agents and authenticators. Additionally, privacy across Relying Parties
0225	is maintained; Relying Parties are not able to detect any properties,
0226	or even the existence, of credentials scoped to other Relying Parties.
0227	
0228	Relying Parties employ the Web Authentication API during two distinct,
0229	but related, ceremonies involving a user. The first is Registration,
0230	where a public key credential is created on an authenticator, and
0231	associated by a Relying Party with the present user's account (the
0232	account may already exist or may be created at this time). The second
0233	is Authentication, where the Relying Party is presented with an
0234	Authentication Assertion proving the presence and consent of the user
0235	who registered the public key credential. Functionally, the Web
0236	Authentication API comprises a PublicKeyCredential which extends the
0237	Credential Management API [CREDENTIAL-MANAGEMENT-1], and infrastructure
0238	which allows those credentials to be used with
0239	navigator.credentials.create() and navigator.credentials.get(). The
0240	former is used during Registration, and the latter during
0241	Authentication.
0242	
0243	Broadly, compliant authenticators protect public key credentials, and
0244	interact with user agents to implement the Web Authentication API. Some
0245	authenticators may run on the same computing device (e.g., smart phone,
0246	tablet, desktop PC) as the user agent is running on. For instance, such
0247	an authenticator might consist of a Trusted Execution Environment (TEE)
0248	applet, a Trusted Platform Module (TPM), or a Secure Element (SE)
0249	integrated into the computing device in conjunction with some means for
0250	user verification, along with appropriate platform software to mediate
0251	access to these components' functionality. Other authenticators may
0252	operate autonomously from the computing device running the user agent,
0253	and be accessed over a transport such as Universal Serial Bus (USB),
0254	Bluetooth Low Energy (BLE) or Near Field Communications (NFC).
0255	
0256	1.1. Use Cases
0257	
0258	The below use case scenarios illustrate use of two very different types
0259	of authenticators, as well as outline further scenarios. Additional
0260	scenarios, including sample code, are given later in 11 Sample
0261	scenarios.
0262	
0263	1.1.1. Registration
0264	
0265	* On a phone:
0266	+ User navigates to example.com in a browser and signs in to an
0267	existing account using whatever method they have been using
0268	(possibly a legacy method such as a password), or creates a
0269	new account.
0270	+ The phone prompts, "Do you want to register this device with
0271	example.com?"
0272	+ User agrees.
0273	+ The phone prompts the user for a previously configured
0274	authorization gesture (PIN, biometric, etc.); the user
0275	provides this.
0276	+ Website shows message, "Registration complete."
0277	
0278	1.1.2. Authentication

0270
0271
0272
0273
0274
0275
0276
0277
0278
0279
0280
0281
0282
0283
0284
0285
0286
0287
0288
0289
0290
0291
0292
0293
0294
0295
0296
0297
0298
0299
0300
0301
0302
0303
0304
0305
0306
0307
0308
0309
0310
0311
0312
0313
0314
0315
0316
0317
0318
0319
0320
0321
0322
0323
0324
0325
0326
0327
0328
0329
0330
0331
0332
0333
0334
0335
0336
0337
0338
0339

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

2. Conformance

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

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

2.1. Dependencies

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

Base64url encoding

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

CBOR

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

CDDL

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

0279
0280
0281
0282
0283
0284
0285
0286
0287
0288
0289
0290
0291
0292
0293
0294
0295
0296
0297
0298
0299
0300
0301
0302
0303
0304
0305
0306
0307
0308
0309
0310
0311
0312
0313
0314
0315
0316
0317
0318
0319
0320
0321
0322
0323
0324
0325
0326
0327
0328
0329
0330
0331
0332
0333
0334
0335
0336
0337
0338
0339
0340
0341
0342
0343
0344
0345
0346
0347
0348

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

2. Conformance

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

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

2.1. Dependencies

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

Base64url encoding

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

CBOR

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

CDDL

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

034C

0341 Credential Management
0342 The API described in this document is an extension of the
0343 Credential concept defined in [CREDENTIAL-MANAGEMENT-1].
0344
0345 DOM
0346 DOMException and the DOMException values used in this
0347 specification are defined in [DOM4].
0348
0349 ECMAScript
0350 %ArrayBuffer% is defined in [ECMAScript].
0351
0352 HTML
0353 The concepts of relevant settings object, origin, opaque origin,
0354 and is a registrable domain suffix of or is equal to are defined
0355 in [HTML52].
0356
0357 Web Cryptography API
0358 The AlgorithmIdentifier type and the method for normalizing an
0359 algorithm are defined in Web Cryptography API
0360 algorithm-dictionary.
0361
0362 Web IDL
0363 Many of the interface definitions and all of the IDL in this
0364 specification depend on [WebIDL-1]. This updated version of the
0365 Web IDL standard adds support for Promises, which are now the
0366 preferred mechanism for asynchronous interaction in all new web
0367 APIs.
0368
0369 The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
0370 "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
0371 document are to be interpreted as described in [RFC2119].
0372
0373 3. Terminology
0374
0375 Assertion
0376 See Authentication Assertion.
0377
0378 Attestation
0379 Generally, a statement that serves to bear witness, confirm, or
0380 authenticate. In the WebAuthn context, attestation is employed
0381 to attest to the provenance of an authenticator and the data it
0382 emits; including, for example: credential IDs, credential key
0383 pairs, signature counters, etc. Attestation information is
0384 conveyed in attestation objects. See also attestation statement
0385 format, and attestation type.
0386
0387 Attestation Certificate
0388 A X.509 Certificate for the attestation key pair used by an
0389 authenticator to attest to its manufacture and capabilities. At
0390 registration time, the authenticator uses the attestation
0391 private key to sign the Relying Party-specific credential public
0392 key (and additional data) that it generates and returns via the
0393 authenticatorMakeCredential operation. Relying Parties use the
0394 attestation public key conveyed in the attestation certificate
0395 to verify the attestation signature. Note that in the case of
0396 self attestation, the authenticator has no distinct attestation
0397 key pair nor attestation certificate, see self attestation for
0398 details.
0399
0400 Authentication
0401 The ceremony where a user, and the user's computing device(s)
0402 (containing at least one authenticator) work in concert to
0403 cryptographically prove to an Relying Party that the user
0404 controls the private key associated with a previously-registered

034E

0350 COSE
0351 CBOR Object Signing and Encryption (COSE) [RFC8152]. The IANA
0352 COSE Algorithms registry established by this specification is
0353 also used.
0354
0355 Credential Management
0356 The API described in this document is an extension of the
0357 Credential concept defined in [CREDENTIAL-MANAGEMENT-1].
0358
0359 DOM
0360 DOMException and the DOMException values used in this
0361 specification are defined in [DOM4].
0362
0363 ECMAScript
0364 %ArrayBuffer% is defined in [ECMAScript].
0365
0366 HTML
0367 The concepts of relevant settings object, origin, opaque origin,
0368 and is a registrable domain suffix of or is equal to are defined
0369 in [HTML52].
0370
0371 Web IDL
0372 Many of the interface definitions and all of the IDL in this
0373 specification depend on [WebIDL-1]. This updated version of the
0374 Web IDL standard adds support for Promises, which are now the
0375 preferred mechanism for asynchronous interaction in all new web
0376 APIs.
0377
0378 The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
0379 "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
0380 document are to be interpreted as described in [RFC2119].
0381
0382 3. Terminology
0383
0384 Assertion
0385 See Authentication Assertion.
0386
0387 Attestation
0388 Generally, attestation is a statement serving to bear witness,
0389 confirm, or authenticate. In the WebAuthn context, attestation
0390 is employed to attest to the provenance of an authenticator and
0391 the data it emits; including, for example: credential IDs,
0392 credential key pairs, signature counters, etc. An attestation
0393 statement is conveyed in an attestation object during
0394 registration. See also 5.3 Attestation and Figure 3.
0395
0396 Attestation Certificate
0397 A X.509 Certificate for the attestation key pair used by an
0398 authenticator to attest to its manufacture and capabilities. At
0399 registration time, the authenticator uses the attestation
0400 private key to sign the Relying Party-specific credential public
0401 key (and additional data) that it generates and returns via the
0402 authenticatorMakeCredential operation. Relying Parties use the
0403 attestation public key conveyed in the attestation certificate
0404 to verify the attestation signature. Note that in the case of
0405 self attestation, the authenticator has no distinct attestation
0406 key pair nor attestation certificate, see self attestation for
0407 details.
0408
0409 Authentication
0410 The ceremony where a user, and the user's computing device(s)
0411 (containing at least one authenticator) work in concert to
0412 cryptographically prove to an Relying Party that the user
0413 controls the credential private key associated with a

0405 public key credential (see Registration). Note that this
0406 includes employing user verification.

0407
0408 Authentication Assertion
0409 The cryptographically signed AuthenticatorAssertionResponse
0410 object returned by an authenticator as the result of a
0411 authenticatorGetAssertion operation.
0412

0413 Authenticator
0414 A cryptographic device used by a WebAuthn Client to (i) generate
0415 a public key credential and register it with a Relying Party,
0416 and (ii) subsequently used to cryptographically sign and return,
0417 in the form of an Authentication Assertion, a challenge and
0418 other data presented by a Relying Party (in concert with the
0419 WebAuthn Client) in order to effect authentication.

0420
0421 Authorization Gesture
0422 An authorization gesture is a physical interaction performed by
0423 a user with an authenticator as part of a ceremony, such as
0424 registration or authentication. By making such an authorization
0425 gesture, a user provides consent for (i.e., authorizes) a
0426 ceremony to proceed. This may involve user verification if the
0427 employed authenticator is capable, or it may involve a simple
0428 test of user presence.

0429
0430 Biometric Recognition
0431 The automated recognition of individuals based on their
0432 biological and behavioral characteristics
0433 [ISOBiometricVocabulary].

0434
0435 Ceremony
0436 The concept of a ceremony [Ceremony] is an extension of the
0437 concept of a network protocol, with human nodes alongside
0438 computer nodes and with communication links that include user
0439 interface(s), human-to-human communication, and transfers of
0440 physical objects that carry data. What is out-of-band to a
0441 protocol is in-band to a ceremony. In this specification,
0442 Registration and Authentication are ceremonies, and an
0443 authorization gesture is often a component of those ceremonies.

0444
0445 Client
0446 See Conforming User Agent.
0447

0448 Conforming User Agent
0449 A user agent implementing, in conjunction with the underlying
0450 platform, the Web Authentication API and algorithms given in
0451 this specification, and handling communication between
0452 authenticators and Relying Parties.

0414 previously-registered public key credential (see Registration).
0415 Note that this typically includes employing a test of user
0416 presence or user verification.

0417
0418 Authentication Assertion
0419 The cryptographically signed AuthenticatorAssertionResponse
0420 object returned by an authenticator as the result of a
0421 authenticatorGetAssertion operation.

0422
0423 Authenticator
0424 A cryptographic device used by a WebAuthn Client to (i) generate
0425 a public key credential and register it with a Relying Party,
0426 and (ii) subsequently used to cryptographically sign and return,
0427 in the form of an Authentication Assertion, a challenge and
0428 other data presented by a Relying Party (in concert with the
0429 WebAuthn Client) in order to effect authentication.

0430
0431 Authorization Gesture
0432 An authorization gesture is a physical interaction performed by
0433 a user with an authenticator as part of a ceremony, such as
0434 registration or authentication. By making such an authorization
0435 gesture, a user provides consent for (i.e., authorizes) a
0436 ceremony to proceed. This may involve user verification if the
0437 employed authenticator is capable, or it may involve a simple
0438 test of user presence.

0439
0440 Biometric Recognition
0441 The automated recognition of individuals based on their
0442 biological and behavioral characteristics
0443 [ISOBiometricVocabulary].

0444
0445 Ceremony
0446 The concept of a ceremony [Ceremony] is an extension of the
0447 concept of a network protocol, with human nodes alongside
0448 computer nodes and with communication links that include user
0449 interface(s), human-to-human communication, and transfers of
0450 physical objects that carry data. What is out-of-band to a
0451 protocol is in-band to a ceremony. In this specification,
0452 Registration and Authentication are ceremonies, and an
0453 authorization gesture is often a component of those ceremonies.

0454
0455 Client
0456 See Conforming User Agent.

0457
0458 Client-Side
0459 This refers in general to the combination of the user's platform
0460 device, user agent, authenticators, and everything gluing it all
0461 together.

0462
0463 Client-side-resident Credential Private Key
0464 A Client-side-resident Credential Private Key is stored either
0465 on the client platform, or in some cases on the authenticator
0466 itself, e.g., in the case of a discrete first-factor roaming
0467 authenticator. Such client-side credential private key storage
0468 has the property that the authenticator is able to select the
0469 credential private key given only an RP ID, possibly with user
0470 assistance (e.g., by providing the user a pick list of
0471 credentials associated with the RP ID). By definition, the
0472 private key is always exclusively controlled by the
0473 Authenticator. In the case of a Client-side-resident Credential
0474 Private Key, the Authenticator might offload storage of wrapped
0475 key material to the client platform, but the client platform is
0476 not expected to offload the key storage to remote entities (e.g.
0477 RP Server).

0478
0479 Conforming User Agent
0480 A user agent implementing, in conjunction with the underlying
0481 platform, the Web Authentication API and algorithms given in
0482 this specification, and handling communication between
0483 authenticators and Relying Parties.

0453
0454
0455
0456
0457
0458
0459
0460
0461
0462

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

0463
0464
0465
0466
0467
0468
0469
0470
0471
0472
0473
0474
0475
0476
0477
0478
0479
0480
0481
0482
0483
0484
0485
0486
0487
0488
0489
0490
0491
0492
0493
0494

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

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

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

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

0495
0496

Public Key Credential

0484
0485
0486
0487
0488
0489
0490
0491
0492
0493
0494
0495
0496
0497
0498
0499
0500
0501
0502
0503
0504
0505
0506
0507
0508
0509
0510
0511
0512
0513
0514
0515
0516
0517
0518
0519
0520
0521
0522
0523
0524
0525
0526
0527
0528
0529
0530
0531
0532
0533
0534
0535
0536
0537
0538
0539
0540
0541
0542
0543
0544
0545
0546
0547
0548
0549
0550
0551
0552
0553

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

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

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

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

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

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

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

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

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

Public Key Credential

0497 Generically, a credential is data one entity presents to another
0498 in order to authenticate the former's **identity** [RFC4949]. A
0499 WebAuthn public key credential is a { identifier, type } pair
0500 identifying authentication information established by the
0501 authenticator and the Relying Party, together, at registration
0502 time. The authentication information consists of an asymmetric
0503 key pair, where the public key portion is returned to the
0504 Relying Party, which stores it in conjunction with the **present**
0505 user's account. The authenticator maps the private key to the
0506 **Relying Party's RP ID and stores it. Subsequently, only that**
0507 **Relying Party, as identified by its RP ID, is able to employ the**
0508 **public key credential in authentication ceremonies, via the**
0509 **get() method. The Relying Party uses its copy of the stored**
0510 **public key to verify the resultant Authentication Assertion.**

0511
0512 **Test of User Presence**
0513 **TUP**
0514 A test of user presence is a simple form of authorization
0515 gesture and technical process where a user interacts with an
0516 authenticator by (typically) simply touching it (other
0517 modalities may also exist), yielding a boolean result. Note that
0518 this does not constitute user verification because **TUP, by**
0519 **definition, is not capable of biometric recognition, nor does it**
0520 **involve the presentation of a shared secret such as a password**
0521 **or PIN.**

0522
0523 **Client-side-resident Credential Private Key**
0524 A Client-side-resident Credential Private Key is stored either
0525 on the client platform, or in some cases on the authenticator
0526 itself, e.g., in the case of a discrete first-factor roaming
0527 authenticator. Such client-side credential private key storage
0528 has the property that the authenticator is able to select the
0529 credential private key given only an RP ID, possibly with user
0530 assistance (e.g., by providing the user a pick list of
0531 credentials associated with the RP ID). By definition, the
0532 private key is always exclusively controlled by the
0533 Authenticator. In the case of a Client-side-resident Credential
0534 Private Key, the Authenticator might offload storage of wrapped
0535 key material to the client platform, but the client platform is
0536 not expected to offload the key storage to remote entities (e.g.
0537 RP Server).

0538
0539 **Client-Side**
0540 This refers in general to the combination of the user's platform
0541 device, user agent, authenticators, and everything gluing it all
0542 together.

0543
0544 **User Consent**
0545 User consent means the user agrees with what they are being
0546 asked, i.e., it encompasses reading and understanding prompts.
0547 An authorization gesture is a ceremony component often employed
0548 to indicate user consent.

0549
0550 **User Verification**
0551 The technical process by which an authenticator locally
0552 authorizes the invocation of the authenticatorMakeCredential and
0553 authenticatorGetAssertion operations. User verification may be
0554 instigated through various authorization gesture modalities; for
0555 example, through a touch plus pin code, password entry, or
0556 biometric recognition (e.g., presenting a fingerprint)
0557 [ISOBiometricVocabulary]. The intent is to be able to
0558 distinguish individual users. Note that invocation of the
0559 authenticatorMakeCredential and authenticatorGetAssertion
0560 operations implies use of key material managed by the
0561 authenticator. Note that for security, user verification and use
0562 of credential private keys must occur within a single logical
0563 security boundary defining the authenticator.

0564

0554 Generically, a credential is data one entity presents to another
0555 in order to authenticate the former **to the latter** [RFC4949]. A
0556 WebAuthn public key credential is a { identifier, type } pair
0557 identifying authentication information established by the
0558 authenticator and the Relying Party, together, at registration
0559 time. The authentication information consists of an asymmetric
0560 key pair, where the public key portion is returned to the
0561 Relying Party, who **then** stores it in conjunction with the
0562 **present** user's account. The authenticator maps the private key
0563 **portion to the Relying Party's RP ID and stores it.**
0564 **Subsequently, only that** Relying Party, as identified by its RP
0565 **ID, is able to employ the public key credential in**
0566 **authentication ceremonies, via the get() method. The Relying**
0567 **Party uses its stored copy of the credential public key to**
0568 **verify the resultant authentication assertion.**

0569
0570 **Test of User Presence**
0571
0572 A test of user presence is a simple form of authorization
0573 gesture and technical process where a user interacts with an
0574 authenticator by (typically) simply touching it (other
0575 modalities may also exist), yielding a boolean result. Note that
0576 this does not constitute user verification because **a user**
0577 **presence test, by definition, is not capable of biometric**
0578 **recognition, nor does it involve the presentation of a shared**
secret such as a password or PIN.

0579
0580
0581 **User Consent**
0582 User consent means the user agrees with what they are being
0583 asked, i.e., it encompasses reading and understanding prompts.
0584 An authorization gesture is a ceremony component often employed
0585 to indicate user consent.

0586
0587 **User Verification**
0588 The technical process by which an authenticator locally
0589 authorizes the invocation of the authenticatorMakeCredential and
0590 authenticatorGetAssertion operations. User verification may be
0591 instigated through various authorization gesture modalities; for
0592 example, through a touch plus pin code, password entry, or
0593 biometric recognition (e.g., presenting a fingerprint)
0594 [ISOBiometricVocabulary]. The intent is to be able to
0595 distinguish individual users. Note that invocation of the
0596 authenticatorMakeCredential and authenticatorGetAssertion
0597 operations implies use of key material managed by the
0598 authenticator. Note that for security, user verification and use
0599 of credential private keys must occur within a single logical
0600 security boundary defining the authenticator.

0601 **User Present**

0565 User Verified

0566 Upon successful completion of a user verification process, the

0567 user is said to be "verified".

0568

0569 WebAuthn Client

0570 Also referred to herein as simply a client. See also Conforming

0571 User Agent.

0572

0573 4. Web Authentication API

0574

0575 This section normatively specifies the API for creating and using

0576 public key credentials. The basic idea is that the credentials belong

0577 to the user and are managed by an authenticator, with which the Relying

0578 Party interacts through the client (consisting of the browser and

0579 underlying OS platform). Scripts can (with the user's consent) request

0580 the browser to create a new credential for future use by the Relying

0581 Party. Scripts can also request the user's permission to perform

0582 authentication operations with an existing credential. All such

0583 operations are performed in the authenticator and are mediated by the

0584 browser and/or platform on the user's behalf. At no point does the

0585 script get access to the credentials themselves; it only gets

0586 information about the credentials in the form of objects.

0587

0588 In addition to the above script interface, the authenticator may

0589 implement (or come with client software that implements) a user

0590 interface for management. Such an interface may be used, for example,

0591 to reset the authenticator to a clean state or to inspect the current

0592 state of the authenticator. In other words, such an interface is

0593 similar to the user interfaces provided by browsers for managing user

0594 state such as history, saved passwords and cookies. Authenticator

0595 management actions such as credential deletion are considered to be the

0596 responsibility of such a user interface and are deliberately omitted

0597 from the API exposed to scripts.

0598

0599 The security properties of this API are provided by the client and the

0600 authenticator working together. The authenticator, which holds and

0601 manages credentials, ensures that all operations are scoped to a

0602 particular origin, and cannot be replayed against a different origin,

0603 by incorporating the origin in its responses. Specifically, as defined

0604 in 5.2 Authenticator operations, the full origin of the requester is

0605 included, and signed over, in the attestation object produced when a

0606 new credential is created as well as in all assertions produced by

0607 WebAuthn credentials.

0608

0609 Additionally, to maintain user privacy and prevent malicious Relying

0610 Parties from probing for the presence of credentials **belonging to other**

0611 **Relying Parties, each credential is also associated with a Relying**

0612 **Party Identifier, or RP ID. This RP ID is provided by the client to the**

0613 **authenticator for all operations, and the authenticator ensures that**

0614 **credentials created by a Relying Party can only be used in operations**

0615 **requested by the same RP ID. Separating the origin from the RP ID in**

0616 **this way allows the API to be used in cases where a single Relying**

0617 **Party maintains multiple origins.**

0618

0619 The client facilitates these security measures by providing **correct**

0620 **origins and RP IDs** to the authenticator for each operation. Since **this**

0621 **is an integral part of the WebAuthn security model, user agents MUST**

0622 **only expose this API to callers in secure contexts.**

0623

0624 The Web Authentication API is defined by the union of the Web IDL

0625 fragments presented in the following sections. A combined IDL listing

0626 is given in the IDL Index.

0627

0628 4.1. PublicKeyCredential Interface

0629

0602 UP

0603 Upon successful completion of a user presence test, the user is

0604 said to be "present".

0605

0606 User Verified

0607 UV

0608 Upon successful completion of a user verification process, the

0609 user is said to be "verified".

0610

0611 WebAuthn Client

0612 Also referred to herein as simply a client. See also Conforming

0613 User Agent.

0614

0615 4. Web Authentication API

0616

0617 This section normatively specifies the API for creating and using

0618 public key credentials. The basic idea is that the credentials belong

0619 to the user and are managed by an authenticator, with which the Relying

0620 Party interacts through the client (consisting of the browser and

0621 underlying OS platform). Scripts can (with the user's consent) request

0622 the browser to create a new credential for future use by the Relying

0623 Party. Scripts can also request the user's permission to perform

0624 authentication operations with an existing credential. All such

0625 operations are performed in the authenticator and are mediated by the

0626 browser and/or platform on the user's behalf. At no point does the

0627 script get access to the credentials themselves; it only gets

0628 information about the credentials in the form of objects.

0629

0630 In addition to the above script interface, the authenticator may

0631 implement (or come with client software that implements) a user

0632 interface for management. Such an interface may be used, for example,

0633 to reset the authenticator to a clean state or to inspect the current

0634 state of the authenticator. In other words, such an interface is

0635 similar to the user interfaces provided by browsers for managing user

0636 state such as history, saved passwords and cookies. Authenticator

0637 management actions such as credential deletion are considered to be the

0638 responsibility of such a user interface and are deliberately omitted

0639 from the API exposed to scripts.

0640

0641 The security properties of this API are provided by the client and the

0642 authenticator working together. The authenticator, which holds and

0643 manages credentials, ensures that all operations are scoped to a

0644 particular origin, and cannot be replayed against a different origin,

0645 by incorporating the origin in its responses. Specifically, as defined

0646 in 5.2 Authenticator operations, the full origin of the requester is

0647 included, and signed over, in the attestation object produced when a

0648 new credential is created as well as in all assertions produced by

0649 WebAuthn credentials.

0650

0651 Additionally, to maintain user privacy and prevent malicious Relying

0652 Parties from probing for the presence of **public key** credentials

0653 **belonging to other Relying Parties, each credential is also associated**

0654 **with a Relying Party Identifier, or RP ID. This RP ID is provided by**

0655 **the client to the authenticator for all operations, and the**

0656 **authenticator ensures that credentials created by a Relying Party can**

0657 **only be used in operations requested by the same RP ID. Separating the**

0658 **origin from the RP ID in this way allows the API to be used in cases**

0659 **where a single Relying Party maintains multiple origins.**

0660

0661 The client facilitates these security measures by providing **the Relying**

0662 **Party's** origin and RP ID to the authenticator for each operation. Since

0663 **this** is an integral part of the WebAuthn security model, user agents

0664 only expose this API to callers in secure contexts.

0665

0666 The Web Authentication API is defined by the union of the Web IDL

0667 fragments presented in the following sections. A combined IDL listing

0668 is given in the IDL Index.

0669

0670 4.1. PublicKeyCredential Interface

0671

0630 The PublicKeyCredential interface inherits from Credential
0631 [CREDENTIAL-MANAGEMENT-1], and contains the attributes that are
0632 returned to the caller when a new credential is created, or a new
0633 assertion is requested.
0634 [SecureContext]
0635 interface PublicKeyCredential : Credential {
0636 readonly attribute ArrayBuffer rawId;
0637 readonly attribute AuthenticatorResponse response;
0638 readonly attribute AuthenticationExtensions clientExtensionResults;
0639 };
0640
0641 id
0642 This attribute is inherited from Credential, though
0643 PublicKeyCredential overrides Credential's getter, instead
0644 returning the base64url encoding of the data contained in the
0645 object's [[identifier]] internal slot.
0646
0647 rawId
0648 This attribute returns the ArrayBuffer contained in the
0649 [[identifier]] internal slot.
0650
0651 response, of type AuthenticatorResponse, readonly
0652 This attribute contains the authenticator's response to the
0653 client's request to either create a public key credential, or
0654 generate an authentication assertion. If the PublicKeyCredential
0655 is created in response to create(), this attribute's value will
0656 be an AuthenticatorAttestationResponse, otherwise, the
0657 PublicKeyCredential was created in response to get(), and this
0658 attribute's value will be an AuthenticatorAssertionResponse.
0659
0660 clientExtensionResults, of type AuthenticationExtensions, readonly
0661 This attribute contains a map containing extension identifier ->
0662 client extension output entries produced by the extension's
0663 client extension processing.
0664
0665 [[type]]
0666 The PublicKeyCredential interface object's [[type]] internal
0667 slot's value is the string "public-key".
0668
0669 Note: This is reflected via the type attribute getter inherited
0670 from Credential.
0671
0672 [[discovery]]
0673 The PublicKeyCredential interface object's [[discovery]]
0674 internal slot's value is "remote".
0675
0676 [[identifier]]
0677 This internal slot contains an identifier for the credential,
0678 chosen by the platform with help from the authenticator. This
0679 identifier is used to look up credentials for use, and is
0680 therefore expected to be globally unique with high probability
0681 across all credentials of the same type, across all
0682 authenticators. This API does not constrain the format or length
0683 of this identifier, except that it must be sufficient for the
0684 platform to uniquely select a key. For example, an authenticator
0685 without on-board storage may create identifiers containing a
0686 credential private key wrapped with a symmetric key that is
0687 burned into the authenticator.
0688
0689 PublicKeyCredential's interface object inherits Credential's
0690 implementation of [[CollectFromCredentialStore]](options) and
0691 [[Store]](credential), and defines its own implementation of
0692 [[DiscoverFromExternalSource]](options) and [[Create]](options).
0693
0694 4.1.1. CredentialRequestOptions Extension
0695
0696 To support obtaining assertions via navigator.credentials.get(), this
0697 document extends the CredentialRequestOptions dictionary as follows:
0698 partial dictionary CredentialRequestOptions {

0672 The PublicKeyCredential interface inherits from Credential
0673 [CREDENTIAL-MANAGEMENT-1], and contains the attributes that are
0674 returned to the caller when a new credential is created, or a new
0675 assertion is requested.
0676 [SecureContext]
0677 interface PublicKeyCredential : Credential {
0678 [SameObject] readonly attribute ArrayBuffer rawId;
0679 [SameObject] readonly attribute AuthenticatorResponse response;
0680 [SameObject] readonly attribute AuthenticationExtensions clientExtensionResults;
0681 };
0682
0683 id
0684 This attribute is inherited from Credential, though
0685 PublicKeyCredential overrides Credential's getter, instead
0686 returning the base64url encoding of the data contained in the
0687 object's [[identifier]] internal slot.
0688
0689 rawId
0690 This attribute returns the ArrayBuffer contained in the
0691 [[identifier]] internal slot.
0692
0693 response, of type AuthenticatorResponse, readonly
0694 This attribute contains the authenticator's response to the
0695 client's request to either create a public key credential, or
0696 generate an authentication assertion. If the PublicKeyCredential
0697 is created in response to create(), this attribute's value will
0698 be an AuthenticatorAttestationResponse, otherwise, the
0699 PublicKeyCredential was created in response to get(), and this
0700 attribute's value will be an AuthenticatorAssertionResponse.
0701
0702 clientExtensionResults, of type AuthenticationExtensions, readonly
0703 This attribute contains a map containing extension identifier ->
0704 client extension output entries produced by the extension's
0705 client extension processing.
0706
0707 [[type]]
0708 The PublicKeyCredential interface object's [[type]] internal
0709 slot's value is the string "public-key".
0710
0711 Note: This is reflected via the type attribute getter inherited
0712 from Credential.
0713
0714 [[discovery]]
0715 The PublicKeyCredential interface object's [[discovery]]
0716 internal slot's value is "remote".
0717
0718 [[identifier]]
0719 This internal slot contains an identifier for the credential,
0720 chosen by the platform with help from the authenticator. This
0721 identifier is used to look up credentials for use, and is
0722 therefore expected to be globally unique with high probability
0723 across all credentials of the same type, across all
0724 authenticators. This API does not constrain the format or length
0725 of this identifier, except that it must be sufficient for the
0726 platform to uniquely select a key. For example, an authenticator
0727 without on-board storage may create identifiers containing a
0728 credential private key wrapped with a symmetric key that is
0729 burned into the authenticator.
0730
0731 PublicKeyCredential's interface object inherits Credential's
0732 implementation of [[CollectFromCredentialStore]](options) and
0733 [[Store]](credential), and defines its own implementation of
0734 [[DiscoverFromExternalSource]](options) and [[Create]](options).
0735
0736 4.1.1. CredentialCreationOptions Extension


```
0695:   PublicKeyCredentialRequestOptions? publicKey;
0700: };
0701:
0702: 4.1.2. CredentialCreationOptions Extension
0703:
0704: To support registration via navigator.credentials.create(), this
0705: document extends the CredentialCreationOptions dictionary as follows:
0706: partial dictionary CredentialCreationOptions {
0707:   MakeCredentialOptions? publicKey;
0708: };
0709:
0710: 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options)
```

```
0711: method
0712:
0713: PublicKeyCredential's interface object's implementation of the
0714: \[[Create]](options) method allows scripts to call
0715: navigator.credentials.create() to request the creation of a new
0716: credential key pair and PublicKeyCredential, managed by an
0717: authenticator. The user agent will prompt the user for consent. On
0718: success, the returned promise will be resolved with a
0719: PublicKeyCredential containing an AuthenticatorAttestationResponse
0720: object.
0721:
0722: Note: This algorithm is synchronous; the Promise resolution/rejection
0723: is taken care of by navigator.credentials.create().
0724:
0725: This method accepts a single argument:
0726:
0727: options
0728:   This argument is a CredentialCreationOptions object whose
0729:   options["publicKey"] member contains a MakeCredentialOptions
0730:   object specifying how the credential is to be made.
0731:
0732: When this method is invoked, the user agent MUST execute the following
0733: algorithm:
0734: 1. Assert: options["publicKey"] is present.
0735: 2. Let options be the value of options["publicKey"].
0736: 3. If any of the name member of options.rp, the name member of
0737: options.user, the displayName member of options.user, or the id
0738: member of options.user are not present, return a TypeError simple
0739: exception.
0740: 4. If the timeout member of options is present, check if its value
0741: lies within a reasonable range as defined by the platform and if
0742: not, correct it to the closest value lying within that range. Set
0743: adjustedTimeout to this adjusted value. If the timeout member of
0744: options is not present, then set adjustedTimeout to a
0745: platform-specific default.
0746: 5. Let global be the PublicKeyCredential interface object's
0747: environment settings object's global object.
0748: 6. Let callerOrigin be the origin specified by this
0749: PublicKeyCredential interface object's relevant settings object. If
0750: callerOrigin is an opaque origin, return a DOMException whose name
0751: is "NotAllowedError", and terminate this algorithm.
0752: 7. If the id member of options.rp is not present, then set rpId to
0753: callerOrigin.
0754: Otherwise:
0755:   1. Let effectiveDomain be the callerOrigin's effective domain.
0756:   2. If effectiveDomain is null, then return a DOMException whose
0757: name is "SecurityError" and terminate this algorithm.
0758:   3. If options.rp.id is not a registrable domain suffix of and is
```

```
0738:
0739: To support registration via navigator.credentials.create(), this
0740: document extends the CredentialCreationOptions dictionary as follows:
0741: partial dictionary CredentialCreationOptions {
0742:   MakePublicKeyCredentialOptions    publicKey;
0743: };
0744:
0745: 4.1.2. CredentialRequestOptions Extension
0746:
0747: To support obtaining assertions via navigator.credentials.get(), this
0748: document extends the CredentialRequestOptions dictionary as follows:
0749: partial dictionary CredentialRequestOptions {
0750:   PublicKeyCredentialRequestOptions    publicKey;
0751: };
0752:
0753: 4.1.3. Create a new credential - PublicKeyCredential's \[[Create]](options)
0754: method
0755:
0756: PublicKeyCredential's interface object's implementation of the
0757: \[[Create]](options) method allows scripts to call
0758: navigator.credentials.create() to request the creation of a new
0759: credential key pair and PublicKeyCredential, managed by an
0760: authenticator. The user agent will prompt the user for consent. On
0761: success, the returned promise will be resolved with a
0762: PublicKeyCredential containing an AuthenticatorAttestationResponse
0763: object.
0764:
0765: Note: This algorithm is synchronous; the Promise resolution/rejection
0766: is handled by navigator.credentials.create().
0767:
0768: This method accepts a single argument:
0769:
0770: options
0771:   This argument is a CredentialCreationOptions object whose
0772:   options.publicKey member contains a
0773:   MakePublicKeyCredentialOptions object specifying the desired
0774: attributes of the to-be-created public key credential.
0775:
0776: When this method is invoked, the user agent MUST execute the following
0777: algorithm:
0778: 1. Assert: options.publicKey is present.
0779: 2. Let options be the value of options.publicKey.
0780: 3. If any of the name member of options.rp, the name member of
0781: options.user, the displayName member of options.user, or the id
0782: member of options.user are not present, return a TypeError simple
0783: exception.
0784: 4. If the timeout member of options is present, check if its value
0785: lies within a reasonable range as defined by the platform and if
0786: not, correct it to the closest value lying within that range. Set
0787: adjustedTimeout to this adjusted value. If the timeout member of
0788: options is not present, then set adjustedTimeout to a
0789: platform-specific default.
0790: 5. Let global be the PublicKeyCredential's interface object's
0791: environment settings object's global object.
0792: 6. Let callerOrigin be the origin specified by this
0793: PublicKeyCredential interface object's relevant settings object. If
0794: callerOrigin is an opaque origin, return a DOMException whose name
0795: is "NotAllowedError", and terminate this algorithm.
0796: 7. Let effectiveDomain be the callerOrigin's effective domain. If
0797: effective domain is not a valid domain, then return a DOMException
0798: whose name is "SecurityError" and terminate this algorithm.
0799: Note: An effective domain may resolve to a host, which can be
0800: represented in various manners, such as domain, ipv4 address, ipv6
0801: address, opaque host, or empty host. Only the domain format of host
0802: is allowed here.
0803: 8. Let rpId be effectiveDomain.
```

not equal to effectiveDomain, return a DOMException whose name is "SecurityError", and terminate this algorithm.

4. Set rpId to options.rp.id.

8. Let normalizedParameters be a new list whose items are pairs of PublicKeyCredentialType and a dictionary type (as returned by normalizing an algorithm).

9. For each current of options.parameters:

1. If current.type does not contain a PublicKeyCredentialType supported by this implementation, then continue.

2. Let normalizedAlgorithm be the result of normalizing an algorithm [WebCryptoAPI], with alg set to current.algorithm and op set to "generateKey". If an error occurs during this procedure, then continue.

3. Append the pair of current.type and normalizedAlgorithm to normalizedParameters.

10. If normalizedParameters is empty and options.parameters is not empty, cancel the timer started in step 2, return a DOMException whose name is "NotSupportedError", and terminate this algorithm.

11. Let clientExtensions be a new map and let authenticatorExtensions be a new map.

12. If the extensions member of options is present, then for each extensionId -> clientExtensionInput of options.extensions:

1. If extensionId is not supported by this client platform or is not a registration extension, then continue.

2. Set clientExtensions[extensionId] to clientExtensionInput.

3. If extensionId is not an authenticator extension, then continue.

4. Let authenticatorExtensionInput be the (CBOR) result of running extensionId's client extension processing algorithm on clientExtensionInput. If the algorithm returned an error, continue.

5. Set authenticatorExtensions[extensionId] to the base64url encoding of authenticatorExtensionInput.

13. Let collectedClientData be a new CollectedClientData instance whose fields are:

challenge
The base64url encoding of options.challenge

origin
The unicode serialization of rpId

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

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

clientExtensions
clientExtensions

authenticatorExtensions
authenticatorExtensions

14. Let clientDataJSON be the JSON-serialized client data constructed from collectedClientData.

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

16. Let currentlyAvailableAuthenticators be a new ordered set consisting of all authenticators available on this platform.

17. Let selectedAuthenticators be a new ordered set.

18. If currentlyAvailableAuthenticators is empty, return a DOMException

9. If options.rp.id is present:

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

2. Set rpId to options.rp.id.

Note: rpId represents the caller's RP ID. The RP ID defaults to being the caller's origin's effective domain unless the caller has explicitly set options.rp.id when calling create().

10. Let credTypesAndPubKeyAlgs be a new list whose items are pairs of PublicKeyCredentialType and a COSEAlgorithmIdentifier.

11. For each current of options.pubKeyCredParams:

1. If current.type does not contain a PublicKeyCredentialType supported by this implementation, then continue.

2. Let alg be current.alg.

3. Append the pair of current.type and alg to credTypesAndPubKeyAlgs.

12. If credTypesAndPubKeyAlgs is empty and options.pubKeyCredParams is not empty, cancel the timer started in step 2, return a DOMException whose name is "NotSupportedError", and terminate this algorithm.

13. Let clientExtensions be a new map and let authenticatorExtensions be a new map.

14. If the extensions member of options is present, then for each extensionId -> clientExtensionInput of options.extensions:

1. If extensionId is not supported by this client platform or is not a registration extension, then continue.

2. Set clientExtensions[extensionId] to clientExtensionInput.

3. If extensionId is not an authenticator extension, then continue.

4. Let authenticatorExtensionInput be the (CBOR) result of running extensionId's client extension processing algorithm on clientExtensionInput. If the algorithm returned an error, continue.

5. Set authenticatorExtensions[extensionId] to the base64url encoding of authenticatorExtensionInput.

15. Let collectedClientData be a new CollectedClientData instance whose fields are:

challenge
The base64url encoding of options.challenge.

origin
The serialization of callerOrigin.

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

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

clientExtensions
clientExtensions

authenticatorExtensions
authenticatorExtensions

16. Let clientDataJSON be the JSON-serialized client data constructed from collectedClientData.

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

18. Let currentlyAvailableAuthenticators be a new ordered set consisting of all authenticators currently available on this platform.

19. Let selectedAuthenticators be a new ordered set.

20. If currentlyAvailableAuthenticators is empty, return a DOMException

0824 whose name is "NotFoundError", and terminate this algorithm.
0825 19. If options.authenticatorSelection is present, iterate through
0826 currentlyAvailableAuthenticators and do the following for each
0827 authenticator:
0828 1. If attachment is present and its value is not equal to
0829 authenticator's attachment modality, continue.
0830 2. If requireResidentKey is set to true and the authenticator is
0831 not capable of storing a Client-Side-Resident Credential
0832 Private Key, continue.
0833 3. Append authenticator to selectedAuthenticators.
0834 20. If selectedAuthenticators is empty, return a DOMException whose
0835 name is "ConstraintError", and terminate this algorithm.
0836 21. Let issuedRequests be a new ordered set.
0837 22. For each authenticator in currentlyAvailableAuthenticators:
0838 1. Let excludeList be a new list.
0839 2. For each credential C in options.excludeList:
0840 1. If C.transports is not empty, and authenticator is
0841 connected over a transport not mentioned in C.transports,
0842 the client MAY continue.
0843 2. Otherwise, Append C to excludeList.
0844 3. In parallel, invoke the authenticatorMakeCredential operation
0845 on authenticator with rpId, clientDataHash, options.rp,
0846 options.user, normalizedParameters, excludeList, and
0847 authenticatorExtensions as parameters.
0848 4. Append authenticator to issuedRequests.
0849 23. Start a timer for adjustedTimeout milliseconds. Then execute the
0850 following steps in parallel. The task source for these tasks is the
0851 dom manipulation task source.
0852 24. While issuedRequests is not empty, perform the following actions
0853 depending upon the adjustedTimeout timer and responses from the
0854 authenticators:
0855 If the adjustedTimeout timer expires,
0856 For each authenticator in issuedRequests invoke the
0857 authenticatorCancel operation on authenticator and remove
0858 authenticator from issuedRequests.
0859 If any authenticator returns a status indicating that the user
0860 cancelled the operation,
0861 1. Remove authenticator from issuedRequests.
0862 2. For each remaining authenticator in issuedRequests invoke
0863 the authenticatorCancel operation on authenticator and
0864 remove it from issuedRequests.
0865 If any authenticator returns an error status,
0866 Remove authenticator from issuedRequests.
0867 If any authenticator indicates success,
0868 1. Remove authenticator from issuedRequests.
0869 2. Let attestationObject be a new ArrayBuffer, created using
0870 global's %ArrayBuffer%, containing the bytes of the value
0871 returned from the successful authenticatorMakeCredential
0872 operation (which is attObj, as defined in 5.3.4
0873 Generating an Attestation Object).
0874 3. Let id be attestationObject.authData.attestation
0875 data.credential ID (see 5.3.1 Attestation data and 5.1
0876 Authenticator data).
0877 4. Let value be a new PublicKeyCredential object associated
0878 with global whose fields are:
0879 [[identifier]]
0880 id
0881 response

0872 whose name is "NotFoundError", and terminate this algorithm.
0873 21. If options.authenticatorSelection is present, iterate through
0874 currentlyAvailableAuthenticators and do the following for each
0875 authenticator:
0876 1. If aa is present and its value is not equal to authenticator's
0877 attachment modality, continue.
0878 2. If rk is set to true and the authenticator is not capable of
0879 storing a Client-Side-Resident Credential Private Key,
0880 continue.
0881 3. If uv is set to true and the authenticator is not capable of
0882 performing user verification, continue.
0883 4. Append authenticator to selectedAuthenticators.
0884 22. If selectedAuthenticators is empty, return a DOMException whose
0885 name is "ConstraintError", and terminate this algorithm.
0886 23. Let issuedRequests be a new ordered set.
0887 24. For each authenticator in currentlyAvailableAuthenticators:
0888 1. Let excludeCredentialDescriptorList be a new list.
0889 2. For each credential descriptor C in
0890 options.excludeCredentials:
0891 1. If C.transports is not empty, and authenticator is
0892 connected over a transport not mentioned in C.transports,
0893 the client MAY continue.
0894 2. Otherwise, Append C to excludeCredentialDescriptorList.
0895 3. In parallel, invoke the authenticatorMakeCredential operation
0896 on authenticator with rpId, clientDataHash, options.rp,
0897 options.user, options.authenticatorSelection.rk,
0898 credTypesAndPubKeyAlgs, excludeCredentialDescriptorList, and
0899 authenticatorExtensions as parameters.
0900 4. Append authenticator to issuedRequests.
0901 25. Start a timer for adjustedTimeout milliseconds. Then execute the
0902 following steps in parallel. The task source for these tasks is the
0903 dom manipulation task source.
0904 26. While issuedRequests is not empty, perform the following actions
0905 depending upon the adjustedTimeout timer and responses from the
0906 authenticators:
0907 If the adjustedTimeout timer expires,
0908 For each authenticator in issuedRequests invoke the
0909 authenticatorCancel operation on authenticator and remove
0910 authenticator from issuedRequests.
0911 If any authenticator returns a status indicating that the user
0912 cancelled the operation,
0913 1. Remove authenticator from issuedRequests.
0914 2. For each remaining authenticator in issuedRequests invoke
0915 the authenticatorCancel operation on authenticator and
0916 remove it from issuedRequests.
0917 If any authenticator returns an error status,
0918 Remove authenticator from issuedRequests.
0919 If any authenticator indicates success,
0920 1. Remove authenticator from issuedRequests.
0921 2. Let attestationObject be a new ArrayBuffer, created using
0922 global's %ArrayBuffer%, containing the bytes of the value
0923 returned from the successful authenticatorMakeCredential
0924 operation (which is attObj, as defined in 5.3.4
0925 Generating an Attestation Object).
0926 3. Let id be attestationObject.authData.attestation
0927 data.credential ID (see 5.3.1 Attestation data and 5.1
0928 Authenticator data).
0929 4. Let value be a new PublicKeyCredential object associated
0930 with global whose fields are:
0931 [[identifier]]
0932 id
0933 response

0890 A new AuthenticatorAttestationResponse object
0891 associated with global whose fields are:
0892
0893 clientDataJSON
0894 A new ArrayBuffer, created using
0895 global's %ArrayBuffer%, containing the
0896 bytes of clientDataJSON.
0897
0898 attestationObject
0899 attestationObject
0900
0901 clientExtensionResults
0902 A new AuthenticationExtensions object
0903 containing the extension identifier -> client
0904 extension output entries created by running
0905 each extension's client extension processing
0906 algorithm to create the client extension
0907 outputs, for each client extension in
0908 clientDataJSON.clientExtensions.
0909
0910 5. For each remaining authenticator in issuedRequests invoke
0911 the authenticatorCancel operation on authenticator and
0912 remove it from issuedRequests.
0913 6. Return value and terminate this algorithm.
0914
0915 25. Return a DOMException whose name is "NotAllowedError".
0916
0917 During the above process, the user agent SHOULD show some UI to the
0918 user to guide them in the process of selecting and authorizing an
0919 authenticator.
0920
0921 4.1.4. Use an existing credential -
0922 PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
0923
0924 The [[DiscoverFromExternalSource]](options) method is used to discover
0925 and use an existing public key credential, with the user's consent. The
0926 script optionally specifies some criteria to indicate what credentials
0927 are acceptable to it. The user agent and/or platform locates
0928 credentials matching the specified criteria, and guides the user to
0929 pick one that the script will be allowed to use. The user may choose
0930 not to provide a credential even if one is present, for example to
0931 maintain privacy.
0932
0933 Note: This algorithm is synchronous; the Promise resolution/rejection
0934 is **taken care of** by navigator.credentials.get().
0935 **This method takes the following parameters:**
0936
0937 options
0938 A CredentialRequestOptions object, containing a challenge that
0939 the selected authenticator is expected to sign to produce the
0940 assertion, and additional options as described in 4.6 Options
0941 for Assertion Generation (dictionary
0942 PublicKeyCredentialRequestOptions)
0943
0944 When this method is invoked, the user agent MUST execute the following
0945 algorithm:
0946 1. Let publicKeyOptions be the value of options.publicKey member.
0947 2. If the timeout member of publicKeyOptions is present, check if its
0948 value lies within a reasonable range as defined by the platform and
0949 if not, correct it to the closest value lying within that range.
0950 Set adjustedTimeout to this adjusted value. If the timeout member
0951 of publicKeyOptions is not present, then set adjustedTimeout to a
0952 platform-specific default.
0953 3. Let global be the PublicKeyCredential's relevant settings object's
0954 environment settings object's global object.
0955 4. Let callerOrigin be the origin of this CredentialsContainer

0942 A new AuthenticatorAttestationResponse object
0943 associated with global whose fields are:
0944
0945 clientDataJSON
0946 A new ArrayBuffer, created using
0947 global's %ArrayBuffer%, containing the
0948 bytes of clientDataJSON.
0949
0950 attestationObject
0951 attestationObject
0952
0953 clientExtensionResults
0954 A new AuthenticationExtensions object
0955 containing the extension identifier -> client
0956 extension output entries created by running
0957 each extension's client extension processing
0958 algorithm to create the client extension
0959 outputs, for each client extension in
0960 clientDataJSON.clientExtensions.
0961
0962 5. For each remaining authenticator in issuedRequests invoke
0963 the authenticatorCancel operation on authenticator and
0964 remove it from issuedRequests.
0965 6. Return value and terminate this algorithm.
0966
0967 27. Return a DOMException whose name is "NotAllowedError".
0968
0969 During the above process, the user agent SHOULD show some UI to the
0970 user to guide them in the process of selecting and authorizing an
0971 authenticator.
0972
0973 4.1.4. Use an existing credential **to make an assertion** -
0974 PublicKeyCredential's [[DiscoverFromExternalSource]](options) method
0975
0976 The [[DiscoverFromExternalSource]](options) method is used to discover
0977 and use an existing public key credential, with the user's consent. The
0978 script optionally specifies some criteria to indicate what credentials
0979 are acceptable to it. The user agent and/or platform locates
0980 credentials matching the specified criteria, and guides the user to
0981 pick one that the script will be allowed to use. The user may choose
0982 not to provide a credential even if one is present, for example to
0983 maintain privacy.
0984
0985 Note: This algorithm is synchronous; the Promise resolution/rejection
0986 is **handled** by navigator.credentials.get().
0987 **This method accepts a single argument:**
0988
0989 options
0990 **This argument is a CredentialRequestOptions object whose**
0991 **options.publicKey member contains a challenge and additional**
0992 **options as described in 4.5 Options for Assertion Generation**
0993 **(dictionary PublicKeyCredentialRequestOptions). The selected**
0994 **authenticator signs the challenge along with other collected**
0995 **data in order to produce an assertion. See 5.2.2 The**
0996 **authenticatorGetAssertion operation.**
0997
0998 When this method is invoked, the user agent MUST execute the following
0999 algorithm:
1000 1. Assert: options.publicKey is present.
1001 2. Let options be the value of options.publicKey.
1002 3. If the timeout member of options is present, check if its value
1003 lies within a reasonable range as defined by the platform and if
1004 not, correct it to the closest value lying within that range. Set
1005 adjustedTimeout to this adjusted value. If the timeout member of
1006 options is not present, then set adjustedTimeout to a
1007 platform-specific default.
1008 4. Let global be the PublicKeyCredential's interface object's
1009 environment settings object's global object.
1010 5. Let callerOrigin be the origin **specified by this**

0956 object's relevant settings object. If callerOrigin is an opaque
0957 origin, return DOMException whose name is "NotAllowedError", and
0958 terminate this algorithm.
0959 5. If the rpId member of publicKeyOptions is not present, then set
0960 rpId to callerOrigin. Otherwise:
0961 1. Let effectiveDomain be the callerOrigin's effective domain.
0962 2. If effectiveDomain is null, then return a DOMException whose
0963 name is "SecurityError" and terminate this algorithm.
0964 3. If rpId is not a registrable domain suffix of and is not equal
0965 to effectiveDomain, return a DOMException whose name is
0966 "SecurityError", and terminate this algorithm.
0967 4. Set rpId to the rpId.
0968 6. Let clientExtensions be a new map and let authenticatorExtensions
be a new map.
0970 7. If the extensions member of publicKeyOptions is present, then for
0971 each extensionId -> clientExtensionInput of
0972 publicKeyOptions.extensions:
0973 1. If extensionId is not supported by this client platform or is
0974 not an authentication extension, then continue.
0975 2. Set clientExtensions[extensionId] to clientExtensionInput.
0976 3. If extensionId is not an authenticator extension, then
0977 continue.
0978 4. Let authenticatorExtensionInput be the (CBOR) result of
0979 running extensionId's client extension processing algorithm on
0980 clientExtensionInput. If the algorithm returned an error,
0981 continue.
0982 5. Set authenticatorExtensions[extensionId] to the base64url
0983 encoding of authenticatorExtensionInput.
0984 8. Let collectedClientData be a new CollectedClientData instance whose
0985 fields are:
0986 challenge
0987 The base64url encoding of publicKeyOptions.challenge
0988 origin
0989 The unicode serialization of rpId
0990 hashAlg
0991 The recognized algorithm name of the hash algorithm
0992 selected by the client for generating the hash of the
0993 serialized client data
0994 tokenBinding
0995 The Token Binding ID associated with callerOrigin, if one
1000 is available.
1001 clientExtensions
1002 clientExtensions
1003 authenticatorExtensions
1004 authenticatorExtensions
1005 9. Let clientDataJSON be the JSON-serialized client data constructed
1006 from collectedClientData.
1007 10. Let clientDataHash be the hash of the serialized client data
1008 represented by clientDataJSON.
1009 11. Let issuedRequests be a new ordered set.
1010 12. If there are no authenticators currently available on this
1011 platform, return a DOMException whose name is "NotFoundError", and
1012 terminate this algorithm.
1013 13. For each authenticator currently available on this platform,

1012 PublicKeyCredential interface object's relevant settings object. If
1013 callerOrigin is an opaque origin, return a DOMException whose name
1014 is "NotAllowedError", and terminate this algorithm.
1015 6. Let effectiveDomain be the callerOrigin's effective domain. If
1016 effective domain is not a valid domain, then return a DOMException
1017 whose name is "SecurityError" and terminate this algorithm.
1018 Note: An effective domain may resolve to a host, which can be
1019 represented in various manners, such as domain, ipv4 address, ipv6
1020 address, opaque host, or empty host. Only the domain format of host
1021 is allowed here.
1022 7. If options.rpId is not present, then set rpId to effectiveDomain.
1023 Otherwise:
1024 1. If options.rpId is not a registrable domain suffix of and is
1025 not equal to effectiveDomain, return a DOMException whose name
1026 is "SecurityError", and terminate this algorithm.
1027 2. Set rpId to options.rpId.
1028 Note: rpId represents the caller's RP ID. The RP ID defaults
1029 to being the caller's origin's effective domain unless the
1030 caller has explicitly set options.rpId when calling get().
1031 8. Let clientExtensions be a new map and let authenticatorExtensions
1032 be a new map.
1033 9. If the extensions member of options is present, then for each
1034 extensionId -> clientExtensionInput of options.extensions:
1035 1. If extensionId is not supported by this client platform or is
1036 not an authentication extension, then continue.
1037 2. Set clientExtensions[extensionId] to clientExtensionInput.
1038 3. If extensionId is not an authenticator extension, then
1039 continue.
1040 4. Let authenticatorExtensionInput be the (CBOR) result of
1041 running extensionId's client extension processing algorithm on
1042 clientExtensionInput. If the algorithm returned an error,
1043 continue.
1044 5. Set authenticatorExtensions[extensionId] to the base64url
1045 encoding of authenticatorExtensionInput.
1046 10. Let collectedClientData be a new CollectedClientData instance whose
1047 fields are:
1048 challenge
1049 The base64url encoding of options.challenge
1050 origin
1051 The serialization of callerOrigin.
1052 hashAlgorithm
1053 The recognized algorithm name of the hash algorithm
1054 selected by the client for generating the hash of the
1055 serialized client data
1056 tokenBindingId
1057 The Token Binding ID associated with callerOrigin, if one
1058 is available.
1059 clientExtensions
1060 clientExtensions
1061 authenticatorExtensions
1062 authenticatorExtensions
1063 11. Let clientDataJSON be the JSON-serialized client data constructed
1064 from collectedClientData.
1065 12. Let clientDataHash be the hash of the serialized client data
1066 represented by clientDataJSON.
1067 13. Let issuedRequests be a new ordered set.
1068 14. If there are no authenticators currently available on this
1069 platform, return a DOMException whose name is "NotFoundError", and
1070 terminate this algorithm.
1071 15. Let authenticator be a platform-specific handle whose value
1072 identifies an authenticator.
1073 16. For each authenticator currently available on this platform,

perform the following steps:

1. Let `credentialList` be a new list.
2. If `publicKeyOptions.allowList` is not empty, execute a platform-specific procedure to determine which, if any, credentials in `publicKeyOptions.allowList` are present on this authenticator by matching with `publicKeyOptions.allowList.id` and `publicKeyOptions.allowList.type`, and set `credentialList` to this filtered list.
3. If `credentialList` is empty then continue.
4. In parallel, for each credential `C` in `credentialList`:
 1. If `C.transports` is not empty, the client SHOULD select one transport from `transports`. Then, using transport, invoke the `authenticatorGetAssertion` operation on authenticator, with `rpId`, `clientDataHash`, `credentialList`, and `authenticatorExtensions` as parameters.
 2. Otherwise, using local configuration knowledge of the appropriate transport to use with authenticator, invoke the `authenticatorGetAssertion` operation on authenticator with `rpId`, `clientDataHash`, `credentialList`, and `clientExtensions` as parameters.
5. Append authenticator to `issuedRequests`.
14. Start a timer for `adjustedTimeout` milliseconds. Then execute the following steps in parallel. The task source for these tasks is the dom manipulation task source.
 15. While `issuedRequests` is not empty, perform the following actions depending upon the `adjustedTimeout` timer and responses from the authenticators:
 - If the `adjustedTimeout` timer expires,
 - For each authenticator in `issuedRequests` invoke the

perform the following steps:

1. Let `allowCredentialDescriptorList` be a new list.
2. If `options.allowCredentials` is not empty, execute a platform-specific procedure to determine which, if any, `public key` credentials described by `options.allowCredentials` are bound to this authenticator, by matching with `rpId`, `options.allowCredentials.id`, and `options.allowCredentials.type`. Set `allowCredentialDescriptorList` to this filtered list.
3. If `allowCredentialDescriptorList` is not empty
 1. Let `distinctTransports` be a new ordered set.
 2. For each credential descriptor `C` in `allowCredentialDescriptorList`, append each value, if any, of `C.transports` to `distinctTransports`.
Note: This will aggregate only distinct values of transports (for this authenticator) in `distinctTransports` due to the properties of ordered sets.
 3. If `distinctTransports` is not empty
The client selects one transport value from `distinctTransports`, possibly incorporating local configuration knowledge of the appropriate transport to use with authenticator in making its selection.
Then, using transport, invoke in parallel the `authenticatorGetAssertion` operation on authenticator, with `rpId`, `clientDataHash`, `allowCredentialDescriptorList`, and `authenticatorExtensions` as parameters.
 - is empty
Using local configuration knowledge of the appropriate transport to use with authenticator, invoke in parallel the `authenticatorGetAssertion` operation on authenticator with `rpId`, `clientDataHash`, `allowCredentialDescriptorList`, and `clientExtensions` as parameters.
 - is empty
Using local configuration knowledge of the appropriate transport to use with authenticator, invoke in parallel the `authenticatorGetAssertion` operation on authenticator with `rpId`, `clientDataHash`, and `clientExtensions` as parameters.
Note: In this case, the Relying Party did not supply a list of acceptable credential descriptors. Thus the authenticator is being asked to exercise any credential it may possess that is bound to the Relying Party, as identified by `rpId`.
4. Append authenticator to `issuedRequests`.
17. Start a timer for `adjustedTimeout` milliseconds. Then execute the following steps in parallel. The task source for these tasks is the dom manipulation task source.
 18. While `issuedRequests` is not empty, perform the following actions depending upon the `adjustedTimeout` timer and responses from the authenticators:
 - If the `adjustedTimeout` timer expires,
 - For each authenticator in `issuedRequests` invoke the

1047 authenticatorCancel operation on authenticator and remove
1048 authenticator from issuedRequests.
1049
1050 If any authenticator returns a status indicating that the user
1051 cancelled the operation,
1052
1053 1. Remove authenticator from issuedRequests.
1054 2. For each remaining authenticator in issuedRequests invoke
1055 the authenticatorCancel operation on authenticator and
1056 remove it from issuedRequests.
1057
1058 If any authenticator returns an error status,
1059 Remove authenticator from issuedRequests.
1060
1061 If any authenticator indicates success,
1062
1063 1. Remove authenticator from issuedRequests.
1064 2. Let value be a new PublicKeyCredential associated with
1065 global whose fields are:
1066
1067 [[identifier]]
1068 A new ArrayBuffer, created using global's
1069 %ArrayBuffer%, containing the bytes of the
1070 credential ID returned from the successful
1071 authenticatorGetAssertion operation, as
1072 defined in [\[#op-get-assertion\]\]](#).
1073
1074 response
1075 A new AuthenticatorAssertionResponse object
1076 associated with global whose fields are:
1077
1078 clientDataJSON
1079 A new ArrayBuffer, created using
1080 global's %ArrayBuffer%, containing the
1081 bytes of clientDataJSON
1082
1083 authenticatorData
1084 A new ArrayBuffer, created using
1085 global's %ArrayBuffer%, containing the
1086 bytes of the returned authenticatorData
1087
1088 signature
1089 A new ArrayBuffer, created using
1090 global's %ArrayBuffer%, containing the
1091 bytes of the returned signature
1092
1093 clientExtensionResults
1094 A new AuthenticationExtensions object
1095 containing the extension identifier -> client
1096 extension output entries created by running
1097 each extension's client extension processing
1098 algorithm to create the client extension
1099 outputs, for each client extension in
1100 clientDataJSON.clientExtensions.
1101
1102 3. For each remaining authenticator in issuedRequests invoke
1103 the authenticatorCancel operation on authenticator and
1104 remove it from issuedRequests.
1105 4. Return value and terminate this algorithm.
1106
1107 16. Return a DOMException whose name is "NotAllowedError".
1108
1109 During the above process, the user agent SHOULD show some UI to the
1110 user to guide them in the process of selecting and authorizing an
1111 authenticator with which to complete the operation.
1112

1151 authenticatorCancel operation on authenticator and remove
1152 authenticator from issuedRequests.
1153
1154 If any authenticator returns a status indicating that the user
1155 cancelled the operation,
1156
1157 1. Remove authenticator from issuedRequests.
1158 2. For each remaining authenticator in issuedRequests invoke
1159 the authenticatorCancel operation on authenticator and
1160 remove it from issuedRequests.
1161
1162 If any authenticator returns an error status,
1163 Remove authenticator from issuedRequests.
1164
1165 If any authenticator indicates success,
1166
1167 1. Remove authenticator from issuedRequests.
1168 2. Let value be a new PublicKeyCredential associated with
1169 global whose fields are:
1170
1171 [[identifier]]
1172 A new ArrayBuffer, created using global's
1173 %ArrayBuffer%, containing the bytes of the
1174 credential ID returned from the successful
1175 authenticatorGetAssertion operation, as
1176 defined in [5.2.2 The](#)
1177 [authenticatorGetAssertion operation](#).
1178
1179 response
1180 A new AuthenticatorAssertionResponse object
1181 associated with global whose fields are:
1182
1183 clientDataJSON
1184 A new ArrayBuffer, created using
1185 global's %ArrayBuffer%, containing the
1186 bytes of clientDataJSON
1187
1188 authenticatorData
1189 A new ArrayBuffer, created using
1190 global's %ArrayBuffer%, containing the
1191 bytes of the returned authenticatorData
1192
1193 signature
1194 A new ArrayBuffer, created using
1195 global's %ArrayBuffer%, containing the
1196 bytes of the returned signature
1197
1198 clientExtensionResults
1199 A new AuthenticationExtensions object
1200 containing the extension identifier -> client
1201 extension output entries created by running
1202 each extension's client extension processing
1203 algorithm to create the client extension
1204 outputs, for each client extension in
1205 clientDataJSON.clientExtensions.
1206
1207 3. For each remaining authenticator in issuedRequests invoke
1208 the authenticatorCancel operation on authenticator and
1209 remove it from issuedRequests.
1210 4. Return value and terminate this algorithm.
1211
1212 19. Return a DOMException whose name is "NotAllowedError".
1213
1214 During the above process, the user agent SHOULD show some UI to the
1215 user to guide them in the process of selecting and authorizing an
1216 authenticator with which to complete the operation.
1217
1218 4.1.5. Platform Authenticator Availability - PublicKeyCredential's
1219 [isPlatformAuthenticatorAvailable\(\)](#) method
1220

1113 4.2. Authenticator Responses (interface AuthenticatorResponse)
1114
1115 Authenticators respond to **Relying Party** requests by returning an object
1116 derived from the AuthenticatorResponse interface:
1117 [SecureContext]
1118 interface AuthenticatorResponse {
1119 readonly attribute ArrayBuffer clientDataJSON;
1120 };
1121
1122 clientDataJSON, of type ArrayBuffer, readonly
1123 This attribute contains a JSON serialization of the client data
1124 passed to the authenticator by the client in its call to either
1125 create() or get().
1126
1127 4.2.1. Information about Public Key Credential (interface
1128 AuthenticatorAttestationResponse)
1129
1130 The AuthenticatorAttestationResponse interface represents the
1131 authenticator's response to a client's request for the creation of a
1132 new public key credential. It contains information about the new
1133 credential that can be used to identify it for later use, and metadata
1134 that can be used by the Relying Party to assess the characteristics of
1135 the credential during registration.
1136 [SecureContext]
1137 interface AuthenticatorAttestationResponse : AuthenticatorResponse {
1138 readonly attribute ArrayBuffer attestationObject;
1139 };
1140
1141 clientDataJSON
1142 This attribute, inherited from AuthenticatorResponse, contains
1143 the JSON-serialized client data (see 5.3 **Credential**)

1221 Relying Parties use this method to determine whether they can create a
1222 new credential using a platform authenticator. Upon invocation, the
1223 client employs a platform-specific procedure to discover available
1224 platform authenticators. If successful, the client then assesses
1225 whether the user is willing to create a credential using one of the
1226 available platform authenticators. This assessment may include various
1227 factors, such as:
1228 * Whether the user is running in private or incognito mode.
1229 * Whether the user has configured the client to not create such
1230 credentials.
1231 * Whether the user has previously expressed an unwillingness to
1232 create a new credential for this Relying Party, either through
1233 configuration or by declining a user interface prompt.
1234 * The user's explicitly stated intentions, determined through user
1235 interaction.
1236
1237 If this assessment is affirmative, the promise is resolved with the
1238 value of True. Otherwise, the promise is resolved with the value of
1239 False. Based on the result, the Relying Party can take further actions
1240 to guide the user to create a credential.
1241
1242 This method has no arguments and returns a boolean value.
1243
1244 If the promise will return False, the client SHOULD wait a fixed period
1245 of time from the invocation of the method before returning False. This
1246 is done so that callers can not distinguish between the case where the
1247 user was unwilling to create a credential using one of the available
1248 platform authenticators and the case where no platform authenticator
1249 exists. Trying to make these cases indistinguishable is done in an
1250 attempt to not provide additional information that could be used for
1251 fingerprinting. A timeout value on the order of 10 minutes is
1252 recommended; this is enough time for successful user interactions to be
1253 performed but short enough that the dangling promise will still be
1254 resolved in a reasonably timely fashion.
1255 [SecureContext]
1256 partial interface PublicKeyCredential {
1257 [Unscopable] Promise < boolean > isPlatformAuthenticatorAvailable();
1258 };
1259
1260 4.2. Authenticator Responses (interface AuthenticatorResponse)
1261
1262 Authenticators respond to **Relying Party** requests by returning an object
1263 derived from the AuthenticatorResponse interface:
1264 [SecureContext]
1265 interface AuthenticatorResponse {
1266 [SameObject] readonly attribute ArrayBuffer clientDataJSON;
1267 };
1268
1269 clientDataJSON, of type ArrayBuffer, readonly
1270 This attribute contains a JSON serialization of the client data
1271 passed to the authenticator by the client in its call to either
1272 create() or get().
1273
1274 4.2.1. Information about Public Key Credential (interface
1275 AuthenticatorAttestationResponse)
1276
1277 The AuthenticatorAttestationResponse interface represents the
1278 authenticator's response to a client's request for the creation of a
1279 new public key credential. It contains information about the new
1280 credential that can be used to identify it for later use, and metadata
1281 that can be used by the Relying Party to assess the characteristics of
1282 the credential during registration.
1283 [SecureContext]
1284 interface AuthenticatorAttestationResponse : AuthenticatorResponse {
1285 [SameObject] readonly attribute ArrayBuffer attestationObject;
1286 };
1287
1288 clientDataJSON
1289 This attribute, inherited from AuthenticatorResponse, contains
1290 the JSON-serialized client data (see 5.3 **Attestation**) passed to

1144 Attestation) passed to the authenticator by the client in order
1145 to generate this credential. The exact JSON serialization must
1146 be preserved, as the hash of the serialized client data has been
1147 computed over it.
1148
1149 attestationObject, of type ArrayBuffer, readonly
1150 This attribute contains an attestation object, which is opaque
1151 to, and cryptographically protected against tampering by, the
1152 client. The attestation object contains both authenticator data
1153 and an attestation statement. The former contains the AAGUID, a
1154 unique credential ID, and the credential public key. The
1155 contents of the attestation statement are determined by the
1156 attestation statement format used by the authenticator. It also
1157 contains any additional information that the Relying Party's
1158 server requires to validate the attestation statement, as well
1159 as to decode and validate the authenticator data along with the
1160 JSON-serialized client data. For more details, see 5.3
1161 **Credential Attestation as well as Figure 3.**
1162
1163 4.2.2. Web Authentication Assertion (interface
1164 AuthenticatorAssertionResponse)
1165
1166 The AuthenticatorAssertionResponse interface represents an
1167 authenticator's response to a client's request for generation of a new
1168 authentication assertion given the Relying Party's challenge and
1169 optional list of credentials it is aware of. This response contains a
1170 cryptographic signature proving possession of the credential private
1171 key, and optionally evidence of user consent to a specific transaction.
1172 [SecureContext]
1173 interface AuthenticatorAssertionResponse : AuthenticatorResponse {
1174 readonly attribute ArrayBuffer authenticatorData;
1175 readonly attribute ArrayBuffer signature;
1176 };
1177
1178 clientDataJSON
1179 This attribute, inherited from AuthenticatorResponse, contains
1180 the JSON-serialized client data (see 4.8.1 Client data used in
1181 WebAuthn signatures (dictionary CollectedClientData)) passed to
1182 the authenticator by the client in order to generate this
1183 assertion. The exact JSON serialization must be preserved, as
1184 the hash of the serialized client data has been computed over
1185 it.
1186
1187 authenticatorData, of type ArrayBuffer, readonly
1188 This attribute contains the authenticator data returned by the
1189 authenticator. See 5.1 Authenticator data.
1190
1191 signature, of type ArrayBuffer, readonly
1192 This attribute contains the raw signature returned from the
1193 authenticator. See 5.2.2 The authenticatorGetAssertion
1194 operation.
1195
1196 4.3. Parameters for Credential Generation (dictionary
1197 PublicKeyCredentialParameters)
1198
1199 dictionary PublicKeyCredentialParameters {
1200 required PublicKeyCredentialType type;
1201 required AlgorithmIdentifier algorithm;
1202 };
1203
1204 This dictionary is used to supply additional parameters when creating a
1205 new credential.
1206
1207 The type member specifies the type of credential to be created.
1208
1209 The algorithm member specifies the cryptographic signature algorithm
1210 with which the newly generated credential will be used, and thus also
1211 the type of asymmetric key pair to be generated, e.g., RSA or Elliptic
1212 Curve.

1291 the authenticator by the client in order to generate this
1292 credential. The exact JSON serialization must be preserved, as
1293 the hash of the serialized client data has been computed over
1294 it.
1295
1296 attestationObject, of type ArrayBuffer, readonly
1297 This attribute contains an attestation object, which is opaque
1298 to, and cryptographically protected against tampering by, the
1299 client. The attestation object contains both authenticator data
1300 and an attestation statement. The former contains the AAGUID, a
1301 unique credential ID, and the credential public key. The
1302 contents of the attestation statement are determined by the
1303 attestation statement format used by the authenticator. It also
1304 contains any additional information that the Relying Party's
1305 server requires to validate the attestation statement, as well
1306 as to decode and validate the authenticator data along with the
1307 JSON-serialized client data. For more details, see 5.3
1308 Attestation, 5.3.4 Generating an Attestation Object, and Figure
1309 3.
1310
1311 4.2.2. Web Authentication Assertion (interface
1312 AuthenticatorAssertionResponse)
1313
1314 The AuthenticatorAssertionResponse interface represents an
1315 authenticator's response to a client's request for generation of a new
1316 authentication assertion given the Relying Party's challenge and
1317 optional list of credentials it is aware of. This response contains a
1318 cryptographic signature proving possession of the credential private
1319 key, and optionally evidence of user consent to a specific transaction.
1320 [SecureContext]
1321 interface AuthenticatorAssertionResponse : AuthenticatorResponse {
1322 [SameObject] readonly attribute ArrayBuffer authenticatorData;
1323 [SameObject] readonly attribute ArrayBuffer signature;
1324 };
1325
1326 clientDataJSON
1327 This attribute, inherited from AuthenticatorResponse, contains
1328 the JSON-serialized client data (see 4.7.1 Client data used in
1329 WebAuthn signatures (dictionary CollectedClientData)) passed to
1330 the authenticator by the client in order to generate this
1331 assertion. The exact JSON serialization must be preserved, as
1332 the hash of the serialized client data has been computed over
1333 it.
1334
1335 authenticatorData, of type ArrayBuffer, readonly
1336 This attribute contains the authenticator data returned by the
1337 authenticator. See 5.1 Authenticator data.
1338
1339 signature, of type ArrayBuffer, readonly
1340 This attribute contains the raw signature returned from the
1341 authenticator. See 5.2.2 The authenticatorGetAssertion
1342 operation.
1343
1344 4.3. Parameters for Credential Generation (dictionary
1345 PublicKeyCredentialParameters)
1346
1347 dictionary PublicKeyCredentialParameters {
1348 required PublicKeyCredentialType type;
1349 required COSEAlgorithmIdentifier alg;
1350 };
1351
1352 This dictionary is used to supply additional parameters when creating a
1353 new credential.
1354
1355 The type member specifies the type of credential to be created.
1356
1357 The alg member specifies the cryptographic signature algorithm with
1358 which the newly generated credential will be used, and thus also the
1359 type of asymmetric key pair to be generated, e.g., RSA or Elliptic
1360 Curve.

1213
1214 4.4. User Account Parameters for Credential Generation (dictionary
1215 PublicKeyCredentialUserEntity)
1216
1217 dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
1218 DOMString displayName;
1219 };
1220
1221 This dictionary is used to supply additional parameters about the user
1222 account when creating a new credential.
1223
1224 The displayName member contains a friendly name for the user account
1225 (e.g., "John P. Smith").
1226
1227 4.5. Options for Credential Creation (dictionary MakeCredentialOptions)
1228
1229 dictionary MakeCredentialOptions {
1230 required PublicKeyCredentialEntity rp;
1231 required PublicKeyCredentialUserEntity user;
1232
1233 required BufferSource challenge;
1234 required sequence<PublicKeyCredentialParameters> parameters;
1235
1236 unsigned long timeout;
1237 sequence<PublicKeyCredentialDescriptor> excludeList;
1238 AuthenticatorSelectionCriteria authenticatorSelection;
1239 AuthenticationExtensions extensions;
1240 };
1241
1242 rp, of type PublicKeyCredentialEntity
1243 This member contains data about the relying party responsible
1244 for the request.
1245
1246 Its value's name member is required, and contains the friendly
1247 name of the relying party (e.g. "Acme Corporation", "Widgets,
1248 Inc.", or "Awesome Site").
1249
1250 Its value's id member specifies the relying party identifier
1251 with which the credential should be associated. If this
1252 identifier is not explicitly set, it will default to the ASCII
1253 serialization of the CredentialsContainer object's relevant
1254 settings object's origin.
1255
1256 user, of type PublicKeyCredentialUserEntity
1257 This member contains data about the user account for which the
1258 relying party is requesting attestation.
1259
1260 Its value's name member is required, and contains a name for the
1261 user account (e.g., "john.p.smith@example.com" or
1262 "+14255551234").
1263
1264 Its value's displayName member is required, and contains a
1265 friendly name for the user account (e.g., "John P. Smith").
1266
1267 Its value's id member is required, and contains an identifier
1268 for the account, specified by the relying party. This is not
1269 meant to be displayed to the user, but is used by the relying
1270 party to control the number of credentials - an authenticator
1271 will never contain more than one credential for a given relying
1272 party under the same id.
1273
1274 challenge, of type BufferSource
1275 This member contains a challenge intended to be used for
1276 generating the newly created credential's attestation object.
1277
1278 parameters, of type sequence<PublicKeyCredentialParameters>
1279 This member contains information about the desired properties of
1280 the credential to be created. The sequence is ordered from most
1281 preferred to least preferred. The platform makes a best-effort

1361
1362 Note: we use "alg" as the latter member name, rather than spelling-out
1363 "algorithm", because it will be serialized into a message to the
1364 authenticator, which may be sent over a low-bandwidth link.
1365
1366 4.4. Options for Credential Creation (dictionary
1367 MakePublicKeyCredentialOptions)
1368
1369 dictionary MakePublicKeyCredentialOptions {
1370 required PublicKeyCredentialEntity rp;
1371 required PublicKeyCredentialUserEntity user;
1372
1373 required BufferSource challenge;
1374 required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
1375
1376 unsigned long timeout;
1377 sequence<PublicKeyCredentialDescriptor> excludeCredentials = [];
1378 AuthenticatorSelectionCriteria authenticatorSelection;
1379 AuthenticationExtensions extensions;
1380 };
1381
1382 rp, of type PublicKeyCredentialEntity
1383 This member contains data about the Relying Party responsible
1384 for the request.
1385
1386 Its value's name member is required, and contains the friendly
1387 name of the Relying Party (e.g. "Acme Corporation", "Widgets,
1388 Inc.", or "Awesome Site").
1389
1390 Its value's id member specifies the relying party identifier
1391 with which the credential should be associated. If omitted, its
1392 value will be the CredentialsContainer object's relevant
1393 settings object's origin's effective domain.
1394
1395 user, of type PublicKeyCredentialUserEntity
1396 This member contains data about the user account for which the
1397 Relying Party is requesting attestation.
1398
1399 Its value's name member is required, and contains a name for the
1400 user account (e.g., "john.p.smith@example.com" or
1401 "+14255551234").
1402
1403 Its value's displayName member is required, and contains a
1404 friendly name for the user account (e.g., "John P. Smith").
1405
1406 Its value's id member is required, and contains an identifier
1407 for the account, specified by the Relying Party. This is not
1408 meant to be displayed to the user, but is used by the Relying
1409 Party to control the number of credentials - an authenticator
1410 will never contain more than one credential for a given Relying
1411 Party under the same id.
1412
1413 challenge, of type BufferSource
1414 This member contains a challenge intended to be used for
1415 generating the newly created credential's attestation object.
1416
1417 pubKeyCredParams, of type sequence<PublicKeyCredentialParameters>
1418 This member contains information about the desired properties of
1419 the credential to be created. The sequence is ordered from most
1420 preferred to least preferred. The platform makes a best-effort

1282 to create the most preferred credential that it can.
1283
1284 timeout, of type unsigned long
1285 This member specifies a time, in milliseconds, that the caller
1286 is willing to wait for the call to complete. This is treated as
1287 a hint, and may be overridden by the platform.
1288
1289 excludeList, of type sequence<PublicKeyCredentialDescriptor>
1290
1291 This member is intended for use by Relying Parties that wish to
1292 limit the creation of multiple credentials for the same account
1293 on a single authenticator. The platform is requested to return
1294 an error if the new credential would be created on an
1295 authenticator that also contains one of the credentials
1296 enumerated in this parameter.
1297
1298 authenticatorSelection, of type AuthenticatorSelectionCriteria
1299 This member is intended for use by Relying Parties that wish to
1300 select the appropriate authenticators to participate in the
1301 create() or get() operation.
1302
1303 extensions, of type AuthenticationExtensions
1304 This member contains additional parameters requesting additional
1305 processing by the client and authenticator. For example, the
1306 caller may request that only authenticators with certain
1307 capabilities be used to create the credential, or that particular
1308 information be returned in the attestation object. Some
1309 extensions are defined in 8 WebAuthn Extensions; consult the
1310 IANA "WebAuthn Extension Identifier" registry established by
1311 [WebAuthn-Registries] for an up-to-date list of registered
1312 WebAuthn Extensions.
1313
1314 4.5.1. Entity Description
1315 The PublicKeyCredentialEntity dictionary describes a user account or a
1316 relying party with which a credential is associated.
1317 dictionary PublicKeyCredentialEntity {
1318 DOMString id;
1319 DOMString name;
1320 USVString icon;
1321 };
1322
1323 id, of type DOMString
1324 A unique identifier for the entity. This will be the ASCII
1325 serialization of an origin for a relying party, and an arbitrary
1326 string specified by the relying party for user accounts.
1327
1328 name, of type DOMString
1329 A human-friendly identifier for the entity. For example, this
1330 could be a company name for a relying party, or a user's name.
1331 This identifier is intended for display.
1332
1333 icon, of type USVString
1334 A serialized URL which resolves to an image associated with the
1335 entity. For example, this could be a user's avatar or a relying
1336 party's logo.
1337
1338 4.5.2. Authenticator Selection Criteria

1421 to create the most preferred credential that it can.
1422
1423 timeout, of type unsigned long
1424 This member specifies a time, in milliseconds, that the caller
1425 is willing to wait for the call to complete. This is treated as
1426 a hint, and may be overridden by the platform.
1427
1428 excludeCredentials, of type sequence<PublicKeyCredentialDescriptor>,
1429 defaulting to None
1430 This member is intended for use by Relying Parties that wish to
1431 limit the creation of multiple credentials for the same account
1432 on a single authenticator. The platform is requested to return
1433 an error if the new credential would be created on an
1434 authenticator that also contains one of the credentials
1435 enumerated in this parameter.
1436
1437 authenticatorSelection, of type AuthenticatorSelectionCriteria
1438 This member is intended for use by Relying Parties that wish to
1439 select the appropriate authenticators to participate in the
1440 create() or get() operation.
1441
1442 extensions, of type AuthenticationExtensions
1443 This member contains additional parameters requesting additional
1444 processing by the client and authenticator. For example, the
1445 caller may request that only authenticators with certain
1446 capabilities be used to create the credential, or that particular
1447 information be returned in the attestation object. Some
1448 extensions are defined in 8 WebAuthn Extensions; consult the
1449 IANA "WebAuthn Extension Identifier" registry established by
1450 [WebAuthn-Registries] for an up-to-date list of registered
1451 WebAuthn Extensions.
1452
1453 4.4.1. Public Key Entity Description (dictionary PublicKeyCredentialEntity)
1454
1455 The PublicKeyCredentialEntity dictionary describes a user account, or a
1456 Relying Party, with which a public key credential is associated.
1457 dictionary PublicKeyCredentialEntity {
1458 DOMString id;
1459 DOMString name;
1460 USVString icon;
1461 };
1462
1463 id, of type DOMString
1464 A unique identifier for the entity. For a relying party entity,
1465 sets the RP ID. For a user account entity, this will be an
1466 arbitrary string specified by the relying party.
1467
1468 name, of type DOMString
1469 A human-friendly identifier for the entity. For example, this
1470 could be a company name for a Relying Party, or a user's name.
1471 This identifier is intended for display.
1472
1473 icon, of type USVString
1474 A serialized URL which resolves to an image associated with the
1475 entity. For example, this could be a user's avatar or a Relying
1476 Party's logo.
1477
1478 4.4.2. User Account Parameters for Credential Generation (dictionary
1479 PublicKeyCredentialUserEntity)
1480
1481 The PublicKeyCredentialUserEntity dictionary is used to supply
1482 additional user account attributes when creating a new credential.
1483 dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
1484 DOMString displayName;
1485 };
1486
1487 displayName, of type DOMString
1488 A friendly name for the user account (e.g., "John P. Smith").
1489
1490 4.4.3. Authenticator Selection Criteria (dictionary

```
1339
1340
1341   Relying Parties may use the AuthenticatorSelectionCriteria dictionary
1342   to specify their requirements regarding authenticator attributes.
1343   dictionary AuthenticatorSelectionCriteria {
1344     Attachment attachment;
1345     boolean requireResidentKey = false;
1346
1347   };
1348
1349   attachment, of type Attachment
1350   If this member is present, eligible authenticators are filtered
1351   to only authenticators attached with the specified 4.5.3
1352   Credential Attachment enumeration (enum Attachment).
1353
1354   requireResidentKey, of type boolean, defaulting to false
1355   This member describes the Relying Parties' requirements
1356   regarding availability of the Client-side-resident Credential
1357   Private Key. If the parameter is set to true, the authenticator
1358   MUST create a Client-side-resident Credential Private Key when
1359   creating a public key credential.
1360
1361   4.5.3. Credential Attachment enumeration (enum Attachment)
1362
1363   enum Attachment {
1364     "platform",
1365     "cross-platform"
1366
1367   };
1368
1369   Clients may communicate with authenticators using a variety of
1370   mechanisms. For example, a client may use a platform-specific API to
1371   communicate with an authenticator which is physically bound to a
1372   platform. On the other hand, a client may use a variety of standardized
1373   cross-platform transport protocols such as Bluetooth (see 4.8.4
1374   Credential Transport enumeration (enum ExternalTransport)) to discover
1375   and communicate with cross-platform attached authenticators. Therefore,
1376   we use Attachment to describe an authenticator's attachment modality.
1377   We define authenticators that are part of the client's platform as
1378   having a platform attachment, and refer to them as platform
1379   authenticators. While those that are reachable via cross-platform
1380   transport protocols are defined as having cross-platform attachment,
1381   and refer to them as roaming authenticators.
1382   * platform attachment - the respective authenticator is attached
1383   using platform-specific transports. Usually, authenticators of this
1384   class are non-removable from the platform.
1385   * cross-platform attachment - the respective authenticator is
1386   attached using cross-platform transports. Authenticators of this
1387   class are removable from, and can "roam" among, client platforms.
1388
1389   This distinction is important because there are use-cases where only
1390   platform authenticators are acceptable to a Relying Party, and
1391   conversely ones where only roaming authenticators are employed. As a
1392   concrete example of the former, a credential on a platform
1393   authenticator may be used by Relying Parties to quickly and
1394   conveniently reauthenticate the user with a minimum of friction, e.g.,
```

```
1491
1492
1493   AuthenticatorSelectionCriteria)
1494
1495   Relying Parties may use the AuthenticatorSelectionCriteria dictionary
1496   to specify their requirements regarding authenticator attributes.
1497   dictionary AuthenticatorSelectionCriteria {
1498     AuthenticatorAttachment aa; // authenticatorAttachment
1499     boolean rk = false; // requireResidentKey
1500     boolean uv = false; // requireUserVerification
1501
1502   };
1503
1504   aa (authenticatorAttachment), of type AuthenticatorAttachment
1505   If this member is present, eligible authenticators are filtered
1506   to only authenticators attached with the specified 4.4.4
1507   Authenticator Attachment enumeration (enum
1508   AuthenticatorAttachment).
1509
1510   rk (requireResidentKey), of type boolean, defaulting to false
1511   This member describes the Relying Parties' requirements
1512   regarding availability of the Client-side-resident Credential
1513   Private Key. If the parameter is set to true, the authenticator
1514   MUST create a Client-side-resident Credential Private Key when
1515   creating a public key credential.
1516
1517   uv (requireUserVerification), of type boolean, defaulting to false
1518   This member describes the Relying Parties' requirements
1519   regarding the authenticator being capable of performing user
1520   verification. If the parameter is set to true, the authenticator
1521   MUST perform user verification when performing the create()
1522   operation and future 4.1.4 Use an existing credential to make
1523   an assertion - PublicKeyCredential's
1524   [[DiscoverFromExternalSource]](options) method operations when
1525   it is requested to verify the credential.
1526
1527   Note: These identifiers are intentionally short, rather than
1528   descriptive, because they will be serialized into a message to the
1529   authenticator, which may be sent over a low-bandwidth link.
1530
1531   4.4.4. Authenticator Attachment enumeration (enum AuthenticatorAttachment)
1532
1533   enum AuthenticatorAttachment {
1534     "plat", // Platform attachment
1535     "xplat" // Cross-platform attachment
1536
1537   };
1538
1539   Clients may communicate with authenticators using a variety of
1540   mechanisms. For example, a client may use a platform-specific API to
1541   communicate with an authenticator which is physically bound to a
1542   platform. On the other hand, a client may use a variety of standardized
1543   cross-platform transport protocols such as Bluetooth (see 4.7.4
1544   Authenticator Transport enumeration (enum AuthenticatorTransport)) to
1545   discover and communicate with cross-platform attached authenticators.
1546   Therefore, we use AuthenticatorAttachment to describe an
1547   authenticator's attachment modality. We define authenticators that are
1548   part of the client's platform as having a platform attachment, and
1549   refer to them as platform authenticators. While those that are
1550   reachable via cross-platform transport protocols are defined as having
1551   cross-platform attachment, and refer to them as roaming authenticators.
1552   * platform attachment - the respective authenticator is attached
1553   using platform-specific transports. Usually, authenticators of this
1554   class are non-removable from the platform.
1555   * cross-platform attachment - the respective authenticator is
1556   attached using cross-platform transports. Authenticators of this
1557   class are removable from, and can "roam" among, client platforms.
1558
1559   This distinction is important because there are use-cases where only
1560   platform authenticators are acceptable to a Relying Party, and
1561   conversely ones where only roaming authenticators are employed. As a
1562   concrete example of the former, a credential on a platform
1563   authenticator may be used by Relying Parties to quickly and
1564   conveniently reauthenticate the user with a minimum of friction, e.g.,
```


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

4.6. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)

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

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

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

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

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

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

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

4.7. Authentication Extensions (typedef AuthenticationExtensions)

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

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

4.8. Supporting Data Structures

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

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

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

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

4.5. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)

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

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

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

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

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

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

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

4.6. Authentication Extensions (typedef AuthenticationExtensions)

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

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

4.7. Supporting Data Structures

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

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

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

JSON. Its structure is defined by the following Web IDL.
dictionary CollectedClientData {
 required DOMString challenge;
 required DOMString origin;
 required DOMString hashAlg;
 DOMString tokenBinding;
 AuthenticationExtensions clientExtensions;
 AuthenticationExtensions authenticatorExtensions;
};

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

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

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

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

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

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

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

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

4.8.2. Credential Type enumeration (enum PublicKeyCredentialType)

enum PublicKeyCredentialType {
 "public-key"
};

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

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

4.8.3. Credential Descriptor (dictionary PublicKeyCredentialDescriptor)

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

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

JSON. Its structure is defined by the following Web IDL.
dictionary CollectedClientData {
 required DOMString challenge;
 required DOMString origin;
 required DOMString hashAlgorithm;
 DOMString tokenBindingId;
 AuthenticationExtensions clientExtensions;
 AuthenticationExtensions authenticatorExtensions;
};

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

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

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

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

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

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

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

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

4.7.2. Credential Type enumeration (enum PublicKeyCredentialType)

enum PublicKeyCredentialType {
 "public-key"
};

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

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

4.7.3. Credential Descriptor (dictionary PublicKeyCredentialDescriptor)

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

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

1532 The type member contains the type of the credential the caller is
1533 referring to.
1534
1535 The id member contains the identifier of the credential that the caller
1536 is referring to.
1537
1538 4.8.4. **Credential** Transport enumeration (enum **ExternalTransport**)
1539
1540 enum Transport {
1541 "usb",
1542 "nfc",
1543 "ble",
1544 };
1545
1546 Authenticators may communicate with Clients using a variety of
1547 transports. This enumeration defines a hint as to how Clients might
1548 communicate with a particular Authenticator in order to obtain an
1549 assertion for a specific credential. Note that these hints represent
1550 the Relying Party's best belief as to how an Authenticator may be
1551 reached. A Relying Party may obtain a list of transports hints from
1552 some attestation statement formats or via some out-of-band mechanism;
1553 it is outside the scope of this specification to define that mechanism.
1554 * usb - the respective Authenticator may be contacted over USB.
1555 * nfc - the respective Authenticator may be contacted over Near Field
1556 Communication (NFC).
1557 * ble - the respective Authenticator may be contacted over Bluetooth
1558 Smart (Bluetooth Low Energy / BLE).
1559
1560 4.8.5. Cryptographic Algorithm Identifier (type AlgorithmIdentifier)
1561
1562 A string or dictionary identifying a cryptographic algorithm and
1563 optionally a set of parameters for that algorithm. This type is defined
1564 in [WebCryptoAPI].

1565
1566 5. WebAuthn Authenticator model
1567
1568 The API defined in this specification implies a specific abstract
1569 functional model for an authenticator. This section describes the
1570 authenticator model.
1571
1572 Client platforms may implement and expose this abstract model in any
1573 way desired. However, the behavior of the client's Web Authentication
1574 API implementation, when operating on the authenticators supported by
1575 that platform, MUST be indistinguishable from the behavior specified in
1576 4 Web Authentication API.
1577
1578 For authenticators, this model defines the logical operations that they
1579 must support, and the data formats that they expose to the client and
1580 the Relying Party. However, it does not define the details of how
1581 authenticators communicate with the client platform, unless they are
1582 required for interoperability with Relying Parties. For instance, this
1583 abstract model does not define protocols for connecting authenticators
1584 to clients over transports such as USB or NFC. Similarly, this abstract
1585 model does not define specific error codes or methods of returning
1586 them; however, it does define error behavior in terms of the needs of
1587 the client. Therefore, specific error codes are mentioned as a means of
1588 showing which error conditions must be distinguishable (or not) from
1589 each other in order to enable a compliant and secure client
1590 implementation.
1591
1592 In this abstract model, the authenticator provides key management and
1593 cryptographic signatures. It may be embedded in the WebAuthn client, or
1594 housed in a separate device entirely. The authenticator may itself
1595 contain a cryptographic module which operates at a higher security
1596 level than the rest of the authenticator. This is particularly
1597 important for authenticators that are embedded in the WebAuthn client,
1598 as in those cases this cryptographic module (which may, for example, be

1701 The type member contains the type of the credential the caller is
1702 referring to.
1703
1704 The id member contains the identifier of the credential that the caller
1705 is referring to.
1706
1707 4.7.4. **Authenticator** Transport enumeration (enum **AuthenticatorTransport**)
1708
1709 enum **Authenticator**Transport {
1710 "usb",
1711 "nfc",
1712 "ble",
1713 };
1714
1715 Authenticators may communicate with Clients using a variety of
1716 transports. This enumeration defines a hint as to how Clients might
1717 communicate with a particular Authenticator in order to obtain an
1718 assertion for a specific credential. Note that these hints represent
1719 the Relying Party's best belief as to how an Authenticator may be
1720 reached. A Relying Party may obtain a list of transports hints from
1721 some attestation statement formats or via some out-of-band mechanism;
1722 it is outside the scope of this specification to define that mechanism.
1723 * usb - the respective Authenticator may be contacted over USB.
1724 * nfc - the respective Authenticator may be contacted over Near Field
1725 Communication (NFC).
1726 * ble - the respective Authenticator may be contacted over Bluetooth
1727 Smart (Bluetooth Low Energy / BLE).
1728
1729 4.7.5. Cryptographic Algorithm Identifier (typedef **COSEAlgorithmIdentifier**)
1730
1731 typedef long **COSEAlgorithmIdentifier**;
1732
1733 A **COSEAlgorithmIdentifier**'s value is a number identifying a
1734 cryptographic algorithm. The algorithm identifiers SHOULD be values
1735 registered in the IANA COSE Algorithms registry [IANA-COSE-ALGS-REG],
1736 for instance, -7 for "ES256" and -257 for "RS256".

1737
1738 5. WebAuthn Authenticator model
1739
1740 The API defined in this specification implies a specific abstract
1741 functional model for an authenticator. This section describes the
1742 authenticator model.
1743
1744 Client platforms may implement and expose this abstract model in any
1745 way desired. However, the behavior of the client's Web Authentication
1746 API implementation, when operating on the authenticators supported by
1747 that platform, MUST be indistinguishable from the behavior specified in
1748 4 Web Authentication API.
1749
1750 For authenticators, this model defines the logical operations that they
1751 must support, and the data formats that they expose to the client and
1752 the Relying Party. However, it does not define the details of how
1753 authenticators communicate with the client platform, unless they are
1754 required for interoperability with Relying Parties. For instance, this
1755 abstract model does not define protocols for connecting authenticators
1756 to clients over transports such as USB or NFC. Similarly, this abstract
1757 model does not define specific error codes or methods of returning
1758 them; however, it does define error behavior in terms of the needs of
1759 the client. Therefore, specific error codes are mentioned as a means of
1760 showing which error conditions must be distinguishable (or not) from
1761 each other in order to enable a compliant and secure client
1762 implementation.
1763
1764 In this abstract model, the authenticator provides key management and
1765 cryptographic signatures. It may be embedded in the WebAuthn client, or
1766 housed in a separate device entirely. The authenticator may itself
1767 contain a cryptographic module which operates at a higher security
1768 level than the rest of the authenticator. This is particularly
1769 important for authenticators that are embedded in the WebAuthn client,
1770 as in those cases this cryptographic module (which may, for example, be

1599 a TPM) could be considered more trustworthy than the rest of the
1600 authenticator.
1601
1602 Each authenticator stores some number of public key credentials. Each
1603 public key credential has an identifier which is unique (or extremely
1604 unlikely to be duplicated) among all public key credentials. Each
1605 credential is also associated with a Relying Party, whose identity is
1606 represented by a Relying Party Identifier (RP ID).
1607
1608 Each authenticator has an AAGUID, which is a 128-bit identifier that
1609 indicates the type (e.g. make and model) of the authenticator. The
1610 AAGUID MUST be chosen by the manufacturer to be identical across all
1611 substantially identical authenticators made by that manufacturer, and
1612 different (with probability 1-2⁻¹²⁸ or greater) from the AAGUIDs of
1613 all other types of authenticators. The RP MAY use the AAGUID to infer
1614 certain properties of the authenticator, such as certification level
1615 and strength of key protection, using information from other sources.
1616
1617 The primary function of the authenticator is to provide WebAuthn
1618 signatures, which are bound to various contextual data. These data are
1619 observed, and added at different levels of the stack as a signature
1620 request passes from the server to the authenticator. In verifying a
1621 signature, the server checks these bindings against expected values.
1622 These contextual bindings are divided in two: Those added by the RP or
1623 the client, referred to as client data; and those added by the
1624 authenticator, referred to as the authenticator data. The authenticator
1625 signs over the client data, but is otherwise not interested in its
1626 contents. To save bandwidth and processing requirements on the
1627 authenticator, the client hashes the client data and sends only the
1628 result to the authenticator. The authenticator signs over the
1629 combination of the hash of the serialized client data, and its own
1630 authenticator data.
1631
1632 The goals of this design can be summarized as follows.
1633 * The scheme for generating signatures should accommodate cases where
1634 the link between the client platform and authenticator is very
1635 limited, in bandwidth and/or latency. Examples include Bluetooth
1636 Low Energy and Near-Field Communication.
1637 * The data processed by the authenticator should be small and easy to
1638 interpret in low-level code. In particular, authenticators should
1639 not have to parse high-level encodings such as JSON.
1640 * Both the client platform and the authenticator should have the
1641 flexibility to add contextual bindings as needed.
1642 * The design aims to reuse as much as possible of existing encoding
1643 formats in order to aid adoption and implementation.
1644
1645 Authenticators produce cryptographic signatures for two distinct
1646 purposes:
1647 1. An attestation signature is produced when a new credential is
1648 created, and provides cryptographic proof of certain properties of
1649 the credential and the authenticator. For instance, an attestation
1650 signature asserts the type of authenticator (as denoted by its
1651 AAGUID) and the public key of the credential. The attestation
1652 signature is signed by an attestation key, which is chosen
1653 depending on the type of attestation desired. For more details on
1654 attestation, see 5.3 Credential Attestation.
1655
1656 2. An assertion signature is produced when the
1657 authenticatorGetAssertion method is invoked. It represents an
1658 assertion by the authenticator that the user has consented to a
1659 specific transaction, such as logging in, or completing a purchase.
1660 Thus, an assertion signature asserts that the authenticator which
1661 possesses a particular credential private key has established, to
1662 the best of its ability, that the human who is requesting this
1663 transaction is the same human who consented to creating that
1664 particular credential. It also provides additional information that
1665 might be useful to the caller, such as the means by which user
1666 consent was provided, and the prompt that was shown to the user by
the authenticator.

1771 a TPM) could be considered more trustworthy than the rest of the
1772 authenticator.
1773
1774 Each authenticator stores some number of public key credentials. Each
1775 public key credential has an identifier which is unique (or extremely
1776 unlikely to be duplicated) among all public key credentials. Each
1777 credential is also associated with a Relying Party, whose identity is
1778 represented by a Relying Party Identifier (RP ID).
1779
1780 Each authenticator has an AAGUID, which is a 128-bit identifier that
1781 indicates the type (e.g. make and model) of the authenticator. The
1782 AAGUID MUST be chosen by the manufacturer to be identical across all
1783 substantially identical authenticators made by that manufacturer, and
1784 different (with probability 1-2⁻¹²⁸ or greater) from the AAGUIDs of
1785 all other types of authenticators. The RP MAY use the AAGUID to infer
1786 certain properties of the authenticator, such as certification level
1787 and strength of key protection, using information from other sources.
1788
1789 The primary function of the authenticator is to provide WebAuthn
1790 signatures, which are bound to various contextual data. These data are
1791 observed, and added at different levels of the stack as a signature
1792 request passes from the server to the authenticator. In verifying a
1793 signature, the server checks these bindings against expected values.
1794 These contextual bindings are divided in two: Those added by the RP or
1795 the client, referred to as client data; and those added by the
1796 authenticator, referred to as the authenticator data. The authenticator
1797 signs over the client data, but is otherwise not interested in its
1798 contents. To save bandwidth and processing requirements on the
1799 authenticator, the client hashes the client data and sends only the
1800 result to the authenticator. The authenticator signs over the
1801 combination of the hash of the serialized client data, and its own
1802 authenticator data.
1803
1804 The goals of this design can be summarized as follows.
1805 * The scheme for generating signatures should accommodate cases where
1806 the link between the client platform and authenticator is very
1807 limited, in bandwidth and/or latency. Examples include Bluetooth
1808 Low Energy and Near-Field Communication.
1809 * The data processed by the authenticator should be small and easy to
1810 interpret in low-level code. In particular, authenticators should
1811 not have to parse high-level encodings such as JSON.
1812 * Both the client platform and the authenticator should have the
1813 flexibility to add contextual bindings as needed.
1814 * The design aims to reuse as much as possible of existing encoding
1815 formats in order to aid adoption and implementation.
1816
1817 Authenticators produce cryptographic signatures for two distinct
1818 purposes:
1819 1. An attestation signature is produced when a new public key
1820 credential is created via an authenticatorMakeCredential operation.
1821 An attestation signature provides cryptographic proof of certain
1822 properties of the the authenticator and the credential. For
1823 instance, an attestation signature asserts the authenticator type
1824 (as denoted by its AAGUID) and the credential public key. The
1825 attestation signature is signed by an attestation private key,
1826 which is chosen depending on the type of attestation desired. For
1827 more details on attestation, see 5.3 Attestation.
1828
1829 2. An assertion signature is produced when the
1830 authenticatorGetAssertion method is invoked. It represents an
1831 assertion by the authenticator that the user has consented to a
1832 specific transaction, such as logging in, or completing a purchase.
1833 Thus, an assertion signature asserts that the authenticator
1834 possessing a particular credential private key has established, to
1835 the best of its ability, that the user requesting this transaction
1836 is the same user who consented to creating that particular public
1837 key credential. It also asserts additional information, termed
1838 client data, that may be useful to the caller, such as the means by
1839 which user consent was provided, and the prompt shown to the user
1840 by the authenticator. The assertion signature format is illustrated
in Figure 2, below.

1667 The formats of these signatures, as well as the procedures for
1668 generating them, are specified below.
1669
1670 5.1. Authenticator data
1671
1672 The authenticator data structure encodes contextual bindings made by
1673 the authenticator. These bindings are controlled by the authenticator
1674 itself, and derive their trust from the Relying Party's assessment of
1675 the security properties of the authenticator. In one extreme case, the
1676 authenticator may be embedded in the client, and its bindings may be no
1677 more trustworthy than the client data. At the other extreme, the
1678 authenticator may be a discrete entity with high-security hardware and
1679 software, connected to the client over a secure channel. In both cases,
1680 the Relying Party receives the authenticator data in the same format,
1681 and uses its knowledge of the authenticator to make trust decisions.
1682
1683 The authenticator data has a compact but extensible encoding. This is
1684 desired since authenticators can be devices with limited capabilities
1685 and low power requirements, with much simpler software stacks than the
1686 client platform components.
1687
1688 The authenticator data structure is a byte array of 37 bytes or more,
1689 as follows.
1690
1691 Length (in bytes) Description
1692 32 SHA-256 hash of the RP ID associated with the credential.
1693 1 Flags (bit 0 is the least significant bit):
1694 * Bit 0: Test of User Presence (TUP) result.
1695 * Bits 1-5: Reserved for future use (RFU).
1696 * Bit 6: Attestation data included (AT). Indicates whether the
1697 authenticator added attestation data.
1698 * Bit 7: Extension data included (ED). Indicates if the authenticator
1699 data has extensions.
1700

1701 4 Signature counter (signCount), 32-bit unsigned big-endian integer.
1702 variable (if present) attestation data (if present). See 5.3.1
1703 Attestation data for details. Its length depends on the length of the
1704 credential public key and credential ID being attested.
1705 variable (if present) Extension-defined authenticator data. This is a
1706 CBOR [RFC7049] map with extension identifiers as keys, and
1707 authenticator extension outputs as values. See 8 WebAuthn Extensions
1708 for details.
1709
1710 The RP ID is originally received from the client when the credential is
1711 created, and again when an assertion is generated. However, it differs
1712 from other client data in some important ways. First, unlike the client
1713 data, the RP ID of a credential does not change between operations but
1714 instead remains the same for the lifetime of that credential. Secondly,
1715 it is validated by the authenticator during the
1716 authenticatorGetAssertion operation, by verifying that the RP ID
1717 associated with the requested credential exactly matches the RP ID
1718 supplied by the client.
1719

1720 The TUP flag SHALL be set if and only if the authenticator detected a
1721 user through an authenticator specific gesture. The RFU bits SHALL be
1722 set to zero.
1723
1724 For attestation signatures, the authenticator MUST set the AT flag and
1725 include the attestation data. For authentication signatures, the AT
1726 flag MUST NOT be set and the attestation data MUST NOT be included.
1727
1728

1841 The formats of these signatures, as well as the procedures for
1842 generating them, are specified below.
1843
1844 5.1. Authenticator data
1845
1846 The authenticator data structure encodes contextual bindings made by
1847 the authenticator. These bindings are controlled by the authenticator
1848 itself, and derive their trust from the Relying Party's assessment of
1849 the security properties of the authenticator. In one extreme case, the
1850 authenticator may be embedded in the client, and its bindings may be no
1851 more trustworthy than the client data. At the other extreme, the
1852 authenticator may be a discrete entity with high-security hardware and
1853 software, connected to the client over a secure channel. In both cases,
1854 the Relying Party receives the authenticator data in the same format,
1855 and uses its knowledge of the authenticator to make trust decisions.
1856
1857 The authenticator data has a compact but extensible encoding. This is
1858 desired since authenticators can be devices with limited capabilities
1859 and low power requirements, with much simpler software stacks than the
1860 client platform components.
1861
1862 The authenticator data structure is a byte array of 37 bytes or more,
1863 as follows.
1864
1865 Length (in bytes) Description
1866 32 SHA-256 hash of the RP ID associated with the credential.
1867 1 Flags (bit 0 is the least significant bit):
1868 * Bit 0: User Present (UP) result.
1869 + 1 means the user is present.
1870 + 0 means the user is not present.
1871 * Bit 1: Reserved for future use (RFU1).
1872 * Bit 2: User Verified (UV) result.
1873 + 1 means the user is verified.
1874 + 0 means the user is not verified.
1875 * Bits 3-5: Reserved for future use (RFU2).
1876 * Bit 6: Attestation data included (AT).
1877 + Indicates whether the authenticator added attestation data.
1878 * Bit 7: Extension data included (ED).
1879 + Indicates if the authenticator data has extensions.
1880
1881 4 Signature counter (signCount), 32-bit unsigned big-endian integer.
1882 variable (if present) attestation data (if present). See 5.3.1
1883 Attestation data for details. Its length depends on the length of the
1884 credential public key and credential ID being attested.
1885 variable (if present) Extension-defined authenticator data. This is a
1886 CBOR [RFC7049] map with extension identifiers as keys, and
1887 authenticator extension outputs as values. See 8 WebAuthn Extensions
1888 for details.
1889
1890 The RP ID is originally received from the client when the credential is
1891 created, and again when an assertion is generated. However, it differs
1892 from other client data in some important ways. First, unlike the client
1893 data, the RP ID of a credential does not change between operations but
1894 instead remains the same for the lifetime of that credential. Secondly,
1895 it is validated by the authenticator during the
1896 authenticatorGetAssertion operation, by verifying that the RP ID
1897 associated with the requested credential exactly matches the RP ID
1898 supplied by the client, and that the RP ID is a registrable domain
1899 suffix of or is equal to the effective domain of the RP's origin's
1900 effective domain.
1901
1902 The UP flag SHALL be set if and only if the authenticator detected a
1903 user through an authenticator specific gesture. The RFU bits SHALL be
1904 set to zero.
1905
1906 For attestation signatures, the authenticator MUST set the AT flag and
1907 include the attestation data. For authentication signatures, the AT
1908 flag MUST NOT be set and the attestation data MUST NOT be included.
1909
1910

1729 If the authenticator does not include any extension data, it MUST set
1730 the ED flag in the first byte to zero, and to one if extension data is
1731 included.

1732
1733 The figure below shows a visual representation of the authenticator
1734 data structure.
1735 [fido-signature-formats-figure1.html] Authenticator data layout.

1736
1737 Note that the authenticator data describes its own length: If the AT
1738 and ED flags are not set, it is always 37 bytes long. The attestation
1739 data (which is only present if the AT flag is set) describes its own
1740 length. If the ED flag is set, then the total length is 37 bytes plus
1741 the length of the attestation data, plus the length of the CBOR map
1742 that follows.

1743
1744 5.2. Authenticator operations

1745
1746 A client must connect to an authenticator in order to invoke any of the
1747 operations of that authenticator. This connection defines an
1748 authenticator session. An authenticator must maintain isolation between
1749 sessions. It may do this by only allowing one session to exist at any
1750 particular time, or by providing more complicated session management.

1751
1752 The following operations can be invoked by the client in an
1753 authenticator session.

1754
1755 5.2.1. The authenticatorMakeCredential operation

1756
1757 This operation must be invoked in an authenticator session which has no
1758 other operations in progress. It takes the following input parameters:

- 1759 * The caller's RP ID, as determined by the user agent and the client.
- 1760 * The hash of the serialized client data, provided by the client.
- 1761 * The relying party's PublicKeyCredentialEntity.
- 1762 * The user account's PublicKeyCredentialEntity.
- 1763 * The PublicKeyCredentialType and cryptographic parameters requested
1764 by the Relying Party, with the cryptographic algorithms normalized
1765 as per the procedure in Web Cryptography API
1766 algorithm-normalization-normalize-an-algorithm.
- 1767 * A list of PublicKeyCredential objects provided by the Relying Party
1768 with the intention that, if any of these are known to the
1769 authenticator, it should not create a new credential.

1770
1771 * Extension data created by the client based on the extensions
1772 requested by the Relying Party.

1773 * The requireResidentKey parameter of the
1774 options.authenticatorSelection dictionary.

1775
1776 When this operation is invoked, the authenticator must perform the
1777 following procedure:

- 1778 * Check if all the supplied parameters are syntactically well-formed
1779 and of the correct length. If not, return an error code equivalent
1780 to UnknownError and terminate the operation.
- 1781 * Check if at least one of the specified combinations of
1782 PublicKeyCredentialType and cryptographic parameters is supported.
1783 If not, return an error code equivalent to NotSupportedError and
1784 terminate the operation.
- 1785 * Check if a credential matching any of the supplied
1786 PublicKeyCredential identifiers is present on this authenticator.
1787 If so, return an error code equivalent to NotAllowedError and
1788 terminate the operation.
- 1789 * If the requireResidentKey flag is set to true and the authenticator
1790 cannot store a Client-side-resident Credential Private Key, return
1791 an error code equivalent to ConstraintError and terminate the
1792 operation.

1793
1794 * Prompt the user for consent to create a new credential. The prompt
1795 for obtaining this consent is shown by the authenticator if it has

1911 If the authenticator does not include any extension data, it MUST set
1912 the ED flag to zero, and to one if extension data is included.

1913
1914 The figure below shows a visual representation of the authenticator
1915 data structure.
1916 [fido-signature-formats-figure1.svg] Authenticator data layout.

1917
1918 Note that the authenticator data describes its own length: If the AT
1919 and ED flags are not set, it is always 37 bytes long. The attestation
1920 data (which is only present if the AT flag is set) describes its own
1921 length. If the ED flag is set, then the total length is 37 bytes plus
1922 the length of the attestation data, plus the length of the CBOR map
1923 that follows.

1924
1925 5.2. Authenticator operations

1926
1927 A client must connect to an authenticator in order to invoke any of the
1928 operations of that authenticator. This connection defines an
1929 authenticator session. An authenticator must maintain isolation between
1930 sessions. It may do this by only allowing one session to exist at any
1931 particular time, or by providing more complicated session management.

1932
1933 The following operations can be invoked by the client in an
1934 authenticator session.

1935
1936 5.2.1. The authenticatorMakeCredential operation

1937
1938 This operation must be invoked in an authenticator session which has no
1939 other operations in progress. It takes the following input parameters:

- 1940 * The caller's RP ID, as determined by the user agent and the client.
- 1941 * The hash of the serialized client data, provided by the client.
- 1942 * The Relying Party's PublicKeyCredentialEntity.
- 1943 * The user account's PublicKeyCredentialUserEntity.
- 1944 * A sequence of pairs of PublicKeyCredentialType and
1945 COSEAlgorithmIdentifier requested by the Relying Party. This
1946 sequence is ordered from most preferred to least preferred. The
1947 platform makes a best-effort to create the most preferred
1948 credential that it can.
- 1949 * An optional list of PublicKeyCredentialDescriptor objects provided
1950 by the Relying Party with the intention that, if any of these are
1951 known to the authenticator, it should not create a new credential.
- 1952 * The rk member of the options.authenticatorSelection dictionary.
- 1953 * The uv member of the options.authenticatorSelection dictionary.
- 1954 * Extension data created by the client based on the extensions
1955 requested by the Relying Party, if any.

1956
1957 When this operation is invoked, the authenticator must perform the
1958 following procedure:

- 1959 * Check if all the supplied parameters are syntactically well-formed
1960 and of the correct length. If not, return an error code equivalent
1961 to "UnknownError" and terminate the operation.
- 1962 * Check if at least one of the specified combinations of
1963 PublicKeyCredentialType and cryptographic parameters is supported.
1964 If not, return an error code equivalent to "NotSupportedError" and
1965 terminate the operation.
- 1966 * Check if a credential matching any of the supplied
1967 PublicKeyCredential identifiers is present on this authenticator.
1968 If so, return an error code equivalent to "NotAllowedError" and
1969 terminate the operation.
- 1970 * If rk is true and the authenticator cannot store a
1971 Client-side-resident Credential Private Key, return an error code
1972 equivalent to "ConstraintError" and terminate the operation.
- 1973 * If uv is true and the authenticator cannot perform user
1974 verification, return an error code equivalent to "ConstraintError"
1975 and terminate the operation.
- 1976 * Prompt the user for consent to create a new credential. The prompt
1977 for obtaining this consent is shown by the authenticator if it has

1794 its own output capability, or by the user agent otherwise. If the
1795 user denies consent, return an error code equivalent to
1796 NotAllowedError and terminate the operation.
1797 * Once user consent has been obtained, generate a new credential
1798 object:
1799 + Generate a set of cryptographic keys using the most preferred
1800 combination of PublicKeyCredentialType and cryptographic
1801 parameters supported by this authenticator.
1802 + Generate an identifier for this credential, such that this
1803 identifier is globally unique with high probability across all
1804 credentials with the same type across all authenticators.
1805 + Associate the credential with the specified RP ID and the
1806 user's account identifier user.id.
1807 + Delete any older credentials with the same RP ID and user.id
1808 that are stored locally in the authenticator.
1809 * If any error occurred while creating the new credential object,
1810 return an error code equivalent to UnknownError and terminate the
1811 operation.
1812 * Process all the supported extensions requested by the client, and
1813 generate the authenticator data with attestation data as specified
1814 in 5.1 Authenticator data. Use this authenticator data and the
1815 hash of the serialized client data to create an attestation object
1816 for the new credential using the procedure specified in 5.3.4
1817 Generating an Attestation Object. For more details on attestation,
1818 see 5.3 Credential Attestation.
1819
1820 On successful completion of this operation, the authenticator returns
1821 the attestation object to the client.
1822
1823 5.2.2. The authenticatorGetAssertion operation
1824
1825 This operation must be invoked in an authenticator session which has no
1826 other operations in progress. It takes the following input parameters:
1827 * The caller's RP ID, as determined by the user agent and the client.
1828 * The hash of the serialized client data, provided by the client.
1829 * A list of credentials acceptable to the Relying Party (possibly
1830 filtered by the client).
1831 * Extension data created by the client based on the extensions
1832 requested by the Relying Party.
1833
1834 When this method is invoked, the authenticator must perform the
1835 following procedure:
1836 * Check if all the supplied parameters are syntactically well-formed
1837 and of the correct length. If not, return an error code equivalent
1838 to UnknownError and terminate the operation.
1839 * If a list of credentials was supplied by the client, filter it by
1840 removing those credentials that are not present on this
1841 authenticator. If no list was supplied, create a list with all
1842 credentials stored for the caller's RP ID (as determined by an
1843 exact match of the RP ID).
1844 * If the previous step resulted in an empty list, return an error
1845 code equivalent to NotAllowedError and terminate the operation.
1846 * Prompt the user to select a credential from among the above list.
1847 Obtain user consent for using this credential. The prompt for
1848 obtaining this consent may be shown by the authenticator if it has
1849 its own output capability, or by the user agent otherwise.
1850 * Process all the supported extensions requested by the client, and
1851 generate the authenticator data without attestation data as
1852 specified in 5.1 Authenticator data. Concatenate this
1853 authenticator data with the hash of the serialized client data to
1854 generate an assertion signature using the private key of the
1855 selected credential as shown below. A simple, undelimited
1856 concatenation is safe to use here because the authenticator data
1857 describes its own length. The hash of the serialized client data
1858 (which potentially has a variable length) is always the last
1859 element.
1860 * If any error occurred while generating the assertion signature,
1861 return an error code equivalent to UnknownError and terminate the
1862 operation.
1863

1978 its own output capability, or by the user agent otherwise. If the
1979 user denies consent, return an error code equivalent to
1980 NotAllowedError and terminate the operation.
1981 * Once user consent has been obtained, generate a new credential
1982 object:
1983 + Generate a set of cryptographic keys using the most preferred
1984 combination of PublicKeyCredentialType and cryptographic
1985 parameters supported by this authenticator.
1986 + Generate an identifier for this credential, such that this
1987 identifier is globally unique with high probability across all
1988 credentials with the same type across all authenticators.
1989 + Associate the credential with the specified RP ID and the
1990 user's account identifier user.id.
1991 + Delete any older credentials with the same RP ID and user.id
1992 that are stored locally by the authenticator.
1993 * If any error occurred while creating the new credential object,
1994 return an error code equivalent to UnknownError and terminate the
1995 operation.
1996 * Process all the supported extensions requested by the client, and
1997 generate the authenticator data with attestation data as specified
1998 in 5.1 Authenticator data. Use this authenticator data and the
1999 hash of the serialized client data to create an attestation object
2000 for the new credential using the procedure specified in 5.3.4
2001 Generating an Attestation Object. For more details on attestation,
2002 see 5.3 Attestation.
2003
2004 On successful completion of this operation, the authenticator returns
2005 the attestation object to the client.
2006
2007 5.2.2. The authenticatorGetAssertion operation
2008
2009 This operation must be invoked in an authenticator session which has no
2010 other operations in progress. It takes the following input parameters:
2011 * The caller's RP ID, as determined by the user agent and the client.
2012 * The hash of the serialized client data, provided by the client.
2013 * A list of credentials acceptable to the Relying Party (possibly
2014 filtered by the client), if any.
2015 * Extension data created by the client based on the extensions
2016 requested by the Relying Party, if any.
2017
2018 When this method is invoked, the authenticator must perform the
2019 following procedure:
2020 * Check if all the supplied parameters are syntactically well-formed
2021 and of the correct length. If not, return an error code equivalent
2022 to UnknownError and terminate the operation.
2023 * If a list of credentials was supplied by the client, filter it by
2024 removing those credentials that are not present on this
2025 authenticator. If no list was supplied, create a list with all
2026 credentials stored for the caller's RP ID (as determined by an
2027 exact match of the RP ID).
2028 * If the previous step resulted in an empty list, return an error
2029 code equivalent to NotAllowedError and terminate the operation.
2030 * Prompt the user to select a credential from among the above list.
2031 Obtain user consent for using this credential. The prompt for
2032 obtaining this consent may be shown by the authenticator if it has
2033 its own output capability, or by the user agent otherwise.
2034 * Process all the supported extensions requested by the client, and
2035 generate the authenticator data as specified in 5.1 Authenticator
2036 data, though without attestation data. Concatenate this
2037 authenticator data with the hash of the serialized client data to
2038 generate an assertion signature using the private key of the
2039 selected credential as shown in Figure 2, below. A simple,
2040 undelimited concatenation is safe to use here because the
2041 authenticator data describes its own length. The hash of the
2042 serialized client data (which potentially has a variable length) is
2043 always the last element.
2044 * If any error occurred while generating the assertion signature,
2045 return an error code equivalent to UnknownError and terminate the
2046 operation.
2047

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

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

- * The identifier of the credential used to generate the signature.
- * The authenticator data used to generate the signature.
- * The assertion signature.

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

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

5.2.3. The authenticatorCancel operation

This operation takes no input parameters and returns no result.

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

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

5.3. Credential Attestation

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

[fido-attestation-structures.html] Relationship of authenticator data and attestation data structures.

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

1. The attestation statement format is the manner in which the signature is represented and the various contextual bindings are incorporated into the attestation statement by the authenticator. In other words, this defines the syntax of the statement. Various

[fido-signature-formats-figure2.svg] Generating an assertion signature.

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

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

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

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

5.2.3. The authenticatorCancel operation

This operation takes no input parameters and returns no result.

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

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

5.3. Attestation

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

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

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

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

1. The attestation statement format is the manner in which the signature is represented and the various contextual bindings are incorporated into the attestation statement by the authenticator. In other words, this defines the syntax of the statement. Various

existing devices and platforms (such as TPMs and the Android OS) have previously defined attestation statement formats. This specification supports a variety of such formats in an extensible way, as defined in 5.3.2 Attestation Statement Formats.

2. The attestation type defines the semantics of the attestation statement and its underlying trust model. It defines how a Relying Party establishes trust in a particular attestation statement, after verifying that it is cryptographically valid. This specification supports a number of attestation types, as described in 5.3.3 Attestation Types.

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

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

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

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

5.3.1. Attestation data

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

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

```
pubKey = $pubKeyFmt
```

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

```
pubKeyFmt /= rsaPubKey
rsaPubKey = { alg: rsaAlgName, n: biguint, e: uint }
rsaAlgName = "RS256" / "RS384" / "RS512" / "PS256" / "PS384" / "PS512"
```

2"

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

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

existing devices and platforms (such as TPMs and the Android OS) have previously defined attestation statement formats. This specification supports a variety of such formats in an extensible way, as defined in 5.3.2 Attestation Statement Formats.

2. The attestation type defines the semantics of attestation statements and their underlying trust models. Specifically, it defines how a Relying Party establishes trust in a particular attestation statement, after verifying that it is cryptographically valid. This specification supports a number of attestation types, as described in 5.3.3 Attestation Types.

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

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

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

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

5.3.1. Attestation data

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

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

[RFC7518] sections 6.3.1.1 and 6.3.1.2.

5.3.2. Attestation Statement Formats

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

- * Its attestation statement format identifier.
- * The set of attestation types supported by the format.
- * The syntax of an attestation statement produced in this format, defined using CDDL for the extension point \$attStmtFormat defined in 5.3.4 Generating an Attestation Object.
- * The procedure for computing an attestation statement in this format given the credential to be attested, the authenticator data structure containing the authenticator data for the attestation, and the hash of the serialized client data.
- * The procedure for verifying an attestation statement, which takes as inputs the authenticator data structure containing the authenticator data claimed to have been used for the attestation and the hash of the serialized client data, and returns either:

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

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

5.3.3. Attestation Types

WebAuthn supports multiple attestation types:

Basic Attestation

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

Self Attestation

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

Privacy CA

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

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

Elliptic Curve based Direct Anonymous Attestation (ECDAAs)

In this case, the Authenticator receives direct anonymous attestation (DAA) credentials from a single DAA-Issuer. These DAA credentials are used along with blinding to sign the

5.3.2. Attestation Statement Formats

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

- * Attestation statement format identifier:
- * Supported attestation types:
- * Syntax: The syntax of an attestation statement produced in this format, defined using [CDDL] for the extension point \$attStmtFormat defined in 5.3.4 Generating an Attestation Object.
- * Signing procedure: The signing procedure for computing an attestation statement in this format given the public key credential to be attested, the authenticator data structure containing the authenticator data for the attestation, and the hash of the serialized client data.
- * Verification procedures: The procedure for verifying an attestation statement, which takes as inputs the authenticator data structure containing the authenticator data claimed to have been used for the attestation and the hash of the serialized client data, and returns either:

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

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

5.3.3. Attestation Types

WebAuthn supports multiple attestation types:

Basic Attestation

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

Self Attestation

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

Privacy CA

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

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

Elliptic Curve based Direct Anonymous Attestation (ECDAAs)

In this case, the Authenticator receives direct anonymous attestation (DAA) credentials from a single DAA-Issuer. These DAA credentials are used along with blinding to sign the

2065 attestation data. The concept of blinding avoids the DAA
2066 credentials being misused as global correlation handle. WebAuthn
2067 supports DAA using elliptic curve cryptography and bilinear
2068 pairings, called ECDAA (see [FIDOEcdaaAlgorithm]) in this
2069 specification. Consequently we denote the DAA-Issuer as
2070 ECDAA-Issuer (see [FIDOEcdaaAlgorithm]).

2071 5.3.4. Generating an Attestation Object

2072 This section specifies the algorithm for generating an attestation
2073 object for any attestation statement format.

2074 In order to construct an attestation object for a given **credential**
2075 using a particular attestation statement format, the **authenticator MUST**
2076 first generate the authenticator data.

2077 The authenticator MUST then run the signing procedure for the desired
2078 attestation statement format with this authenticator data and the hash
2079 of the serialized client data as input, and use this to construct an
2080 attestation statement in that attestation statement format.

2081 Finally, the authenticator MUST construct the attestation object as a
2082 CBOR map with the following syntax:

```
2083 attObj = {  
2084   authData: bytes,  
2085   $$attStmtType  
2086 }
```

```
2087 attStmtTemplate = (  
2088   fmt: text,  
2089   attStmt: bytes  
2090 )
```

2091 ; Every attestation statement format must have the above fields
2092 attStmtTemplate .within \$\$attStmtType

2093 The semantics of the fields in the attestation object are as follows:

2094 **fmt**
2095 The attestation statement format identifier associated with the
2096 attestation statement. Each attestation statement format defines
2097 its identifier.

2098 **authData**
2099 The authenticator data used to generate the attestation
2100 statement.

2101 **attStmt**
2102 The attestation statement constructed above. The syntax of this
2103 is defined by the attestation statement format used.

2104 5.3.5. Security Considerations

2105 5.3.5.1. Privacy

2106 Attestation keys may be used to track users or link various online
2107 identities of the same user together. This may be mitigated in several
2108 ways, including:

- 2109 * A WebAuthn authenticator manufacturer may choose to ship all of
2110 their devices with the same (or a fixed number of) attestation
2111 key(s) (called Basic Attestation). This will anonymize the user at
2112 the risk of not being able to revoke a particular attestation key
2113 should its WebAuthn Authenticator be compromised.
- 2114 * A WebAuthn Authenticator may be capable of dynamically generating
2115 different attestation keys (and requesting related certificates)
2116 per origin (following the Privacy CA approach). For example, a
2117 WebAuthn Authenticator can ship with a master attestation key (and
2118 certificate), and combined with a cloud operated privacy CA, can
2119 dynamically generate per origin attestation keys and attestation
2120 certificates.

2235 attestation data. The concept of blinding avoids the DAA
2236 credentials being misused as global correlation handle. WebAuthn
2237 supports DAA using elliptic curve cryptography and bilinear
2238 pairings, called ECDAA (see [FIDOEcdaaAlgorithm]) in this
2239 specification. Consequently we denote the DAA-Issuer as
2240 ECDAA-Issuer (see [FIDOEcdaaAlgorithm]).

2241 5.3.4. Generating an Attestation Object

2242 This section specifies the algorithm for generating an attestation
2243 object (see: Figure 3) for any attestation statement format.

2244 In order to construct an attestation object for a given **public key**
2245 **credential** using a particular attestation statement format, the
2246 **authenticator MUST** first generate the authenticator data.

2247 The authenticator MUST then run the signing procedure for the desired
2248 attestation statement format with this authenticator data and the hash
2249 of the serialized client data as input, and use this to construct an
2250 attestation statement in that attestation statement format.

2251 Finally, the authenticator MUST construct the attestation object as a
2252 CBOR map with the following syntax:

```
2253 attObj = {  
2254   authData: bytes,  
2255   $$attStmtType  
2256 }
```

```
2257 attStmtTemplate = (  
2258   fmt: text,  
2259   attStmt: bytes  
2260 )
```

2261 ; Every attestation statement format must have the above fields
2262 attStmtTemplate .within \$\$attStmtType

2263 The semantics of the fields in the attestation object are as follows:

2264 **fmt**
2265 The attestation statement format identifier associated with the
2266 attestation statement. Each attestation statement format defines
2267 its identifier.

2268 **authData**
2269 The authenticator data used to generate the attestation
2270 statement.

2271 **attStmt**
2272 The attestation statement constructed above. The syntax of this
2273 is defined by the attestation statement format used.

2274 5.3.5. Security Considerations

2275 5.3.5.1. Privacy

2276 Attestation keys may be used to track users or link various online
2277 identities of the same user together. This may be mitigated in several
2278 ways, including:

- 2279 * A WebAuthn authenticator manufacturer may choose to ship all of
2280 their devices with the same (or a fixed number of) attestation
2281 key(s) (called Basic Attestation). This will anonymize the user at
2282 the risk of not being able to revoke a particular attestation key
2283 should its WebAuthn Authenticator be compromised.
- 2284 * A WebAuthn Authenticator may be capable of dynamically generating
2285 different attestation keys (and requesting related certificates)
2286 per origin (following the Privacy CA approach). For example, a
2287 WebAuthn Authenticator can ship with a master attestation key (and
2288 certificate), and combined with a cloud operated privacy CA, can
2289 dynamically generate per origin attestation keys and attestation
2290 certificates.

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

5.3.5.2. Attestation Certificate and Attestation Certificate CA Compromise

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

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

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

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

5.3.5.3. Attestation Certificate Hierarchy

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

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

6. Relying Party Operations

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

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

5.3.5.2. Attestation Certificate and Attestation Certificate CA Compromise

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

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

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

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

5.3.5.3. Attestation Certificate Hierarchy

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

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

6. Relying Party Operations

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

operations that the Relying Party must perform upon receipt of these structures.

6.1. Registering a new credential

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

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

operations that the Relying Party must perform upon receipt of these structures.

6.1. Registering a new credential

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

1. Perform JSON deserialization on the `clientDataJSON` field of the `AuthenticatorAttestationResponse` object to extract the client data `C` claimed as collected during the credential creation.
2. Verify that the challenge in `C` matches the challenge that was sent to the authenticator in the `create()` call.
3. Verify that the origin in `C` matches the Relying Party's origin.
4. Verify that the `tokenBindingId` in `C` matches the Token Binding ID for the TLS connection over which the attestation was obtained.
5. Verify that the `clientExtensions` in `C` is a proper subset of the extensions requested by the RP and that the `authenticatorExtensions` in `C` is also a proper subset of the extensions requested by the RP.
6. Compute the hash of `clientDataJSON` using the algorithm identified by `C.hashAlgorithm`.
7. Perform CBOR decoding on the `attestationObject` field of the `AuthenticatorAttestationResponse` structure to obtain the attestation statement format `fmt`, the authenticator data `authData`, and the attestation statement `attStmt`.
8. Verify that the RP ID hash in `authData` is indeed the SHA-256 hash of the RP ID expected by the RP.
9. Determine the attestation statement format by performing an USASCII case-sensitive match on `fmt` against the set of supported WebAuthn Attestation Statement Format Identifier values. The up-to-date list of registered WebAuthn Attestation Statement Format Identifier values is maintained in the in the IANA registry of the same name [WebAuthn-Registries].
10. Verify that `attStmt` is a correct, validly-signed attestation statement, using the attestation statement format `fmt`'s verification procedure given authenticator data `authData` and the hash of the serialized client data computed in step 6.
11. If validation is successful, obtain a list of acceptable trust anchors (attestation root certificates or ECDAA-Issuer public keys) for that attestation type and attestation statement format `fmt`, from a trusted source or from policy. For example, the FIDO Metadata Service [FIDOMetadataService] provides one way to obtain such information, using the AAGUID in the attestation data contained in `authData`.
12. Assess the attestation trustworthiness using the outputs of the verification procedure in step 10, as follows:
 - + If self attestation was used, check if self attestation is acceptable under Relying Party policy.
 - + If ECDAA was used, verify that the identifier of the ECDAA-Issuer public key used is included in the set of acceptable trust anchors obtained in step 11.
 - + Otherwise, use the X.509 certificates returned by the verification procedure to verify that the attestation public key correctly chains up to an acceptable root certificate.
13. If the attestation statement `attStmt` verified successfully and is found to be trustworthy, then register the new credential with the account that was denoted in the `options.user` passed to `create()`, by associating it with the credential ID and credential public key contained in `authData`'s attestation data, as appropriate for the Relying Party's systems.
14. If the attestation statement `attStmt` successfully verified but is not trustworthy per step 12 above, the Relying Party SHOULD fail the registration ceremony.
NOTE: However, if permitted by policy, the Relying Party MAY register the credential ID and credential public key but treat the credential as one with self attestation (see 5.3.3 Attestation Types). If doing so, the Relying Party is asserting there is no cryptographic proof that the public key credential has been generated by a particular authenticator model. See [FIDOSecRef] and [UAFProtocol] for a more detailed discussion.

2275 15. If verification of the attestation statement failed, the Relying
2276 Party MUST fail the registration ceremony.
2277
2278 Verification of attestation objects requires that the Relying Party has
2279 a trusted method of determining acceptable trust anchors in step 11
2280 above. Also, if certificates are being used, the Relying Party must
2281 have access to certificate status information for the intermediate CA
2282 certificates. The Relying Party must also be able to build the
2283 attestation certificate chain if the client did not provide this chain
2284 in the attestation information.
2285
2286 To avoid ambiguity during authentication, the Relying Party SHOULD
2287 check that each credential is registered to no more than one user. If
2288 registration is requested for a credential that is already registered
2289 to a different user, the Relying Party SHOULD fail this ceremony, or it
2290 MAY decide to accept the registration, e.g. while deleting the older
2291 registration.
2292
2293 6.2. Verifying an authentication assertion
2294
2295 When verifying a given PublicKeyCredential structure (credential) as
2296 part of an authentication ceremony, the Relying Party MUST proceed as
2297 follows:
2298 1. Using credential's id attribute (or the corresponding rawId, if
2299 base64url encoding is inappropriate for your use case), look up the
2300 corresponding credential public key.
2301 2. Let cData, aData and sig denote the value of credential's
2302 response's clientDataJSON, authenticatorData, and signature
2303 respectively.
2304 3. Perform JSON deserialization on cData to extract the client data C
2305 used for the signature.
2306 4. Verify that the challenge member of C matches the challenge that
2307 was sent to the authenticator in the
2308 PublicKeyCredentialRequestOptions passed to the get() call.
2309 5. Verify that the origin member of C matches the Relying Party's
2310 origin.
2311 6. Verify that the tokenBinding member of C (if present) matches the
2312 Token Binding ID for the TLS connection over which the signature
2313 was obtained.
2314 7. Verify that the clientExtensions member of C is a proper subset of
2315 the extensions requested by the Relying Party and that the
2316 authenticatorExtensions in C is also a proper subset of the
2317 extensions requested by the Relying Party.
2318 8. Verify that the RP ID hash in aData is the SHA-256 hash of the RP
2319 ID expected by the Relying Party.
2320 9. Let hash be the result of computing a hash over the cData using the
2321 algorithm represented by the hashAlg member of C.
2322 10. Using the credential public key looked up in step 1, verify that
2323 sig is a valid signature over the binary concatenation of aData and
2324 hash.
2325 11. If all the above steps are successful, continue with the
2326 authentication ceremony as appropriate. Otherwise, fail the
2327 authentication ceremony.
2328
2329 7. Defined Attestation Statement Formats
2330
2331 WebAuthn supports pluggable attestation statement formats. This section
2332 defines an initial set of such formats.
2333
2334 7.1. Attestation Statement Format Identifiers
2335
2336 Attestation statement formats are identified by a string, called a
2337 attestation statement format identifier, chosen by the author of the
2338 attestation statement format.
2339
2340 Attestation statement format identifiers SHOULD be registered per
2341 [WebAuthn-Registries] "Registries for Web Authentication (WebAuthn)".
2342 All registered attestation statement format identifiers are unique
2343 amongst themselves as a matter of course.
2344

2445 15. If verification of the attestation statement failed, the Relying
2446 Party MUST fail the registration ceremony.
2447
2448 Verification of attestation objects requires that the Relying Party has
2449 a trusted method of determining acceptable trust anchors in step 11
2450 above. Also, if certificates are being used, the Relying Party must
2451 have access to certificate status information for the intermediate CA
2452 certificates. The Relying Party must also be able to build the
2453 attestation certificate chain if the client did not provide this chain
2454 in the attestation information.
2455
2456 To avoid ambiguity during authentication, the Relying Party SHOULD
2457 check that each credential is registered to no more than one user. If
2458 registration is requested for a credential that is already registered
2459 to a different user, the Relying Party SHOULD fail this ceremony, or it
2460 MAY decide to accept the registration, e.g. while deleting the older
2461 registration.
2462
2463 6.2. Verifying an authentication assertion
2464
2465 When verifying a given PublicKeyCredential structure (credential) as
2466 part of an authentication ceremony, the Relying Party MUST proceed as
2467 follows:
2468 1. Using credential's id attribute (or the corresponding rawId, if
2469 base64url encoding is inappropriate for your use case), look up the
2470 corresponding credential public key.
2471 2. Let cData, aData and sig denote the value of credential's
2472 response's clientDataJSON, authenticatorData, and signature
2473 respectively.
2474 3. Perform JSON deserialization on cData to extract the client data C
2475 used for the signature.
2476 4. Verify that the challenge member of C matches the challenge that
2477 was sent to the authenticator in the
2478 PublicKeyCredentialRequestOptions passed to the get() call.
2479 5. Verify that the origin member of C matches the Relying Party's
2480 origin.
2481 6. Verify that the tokenBindingId member of C (if present) matches the
2482 Token Binding ID for the TLS connection over which the signature
2483 was obtained.
2484 7. Verify that the clientExtensions member of C is a proper subset of
2485 the extensions requested by the Relying Party and that the
2486 authenticatorExtensions in C is also a proper subset of the
2487 extensions requested by the Relying Party.
2488 8. Verify that the RP ID hash in aData is the SHA-256 hash of the RP
2489 ID expected by the Relying Party.
2490 9. Let hash be the result of computing a hash over the cData using the
2491 algorithm represented by the hashAlgorithm member of C.
2492 10. Using the credential public key looked up in step 1, verify that
2493 sig is a valid signature over the binary concatenation of aData and
2494 hash.
2495 11. If all the above steps are successful, continue with the
2496 authentication ceremony as appropriate. Otherwise, fail the
2497 authentication ceremony.
2498
2499 7. Defined Attestation Statement Formats
2500
2501 WebAuthn supports pluggable attestation statement formats. This section
2502 defines an initial set of such formats.
2503
2504 7.1. Attestation Statement Format Identifiers
2505
2506 Attestation statement formats are identified by a string, called a
2507 attestation statement format identifier, chosen by the author of the
2508 attestation statement format.
2509
2510 Attestation statement format identifiers SHOULD be registered per
2511 [WebAuthn-Registries] "Registries for Web Authentication (WebAuthn)".
2512 All registered attestation statement format identifiers are unique
2513 amongst themselves as a matter of course.
2514

2345	Unregistered attestation statement format identifiers SHOULD use
2346	lowercase reverse domain-name naming, using a domain name registered by
2347	the developer, in order to assure uniqueness of the identifier. All
2348	attestation statement format identifiers MUST be a maximum of 32 octets
2349	in length and MUST consist only of printable USASCII characters,
2350	excluding backslash and doublequote, i.e., VCHAR as defined in
2351	[RFC5234] but without %x22 and %x5c. (Note: This means attestation
2352	statement format identifiers based on domain names MUST incorporate
2353	only LDH Labels [RFC5890].) Implementations MUST match WebAuthn
2354	attestation statement format identifiers in a case-sensitive fashion.
2355	
2356	Attestation statement formats that may exist in multiple versions
2357	SHOULD include a version in their identifier. In effect, different
2358	versions are thus treated as different formats, e.g., packed2 as a new
2359	version of the packed attestation statement format.
2360	
2361	The following sections present a set of currently-defined and
2362	registered attestation statement formats and their identifiers. The
2363	up-to-date list of registered WebAuthn Extensions is maintained in the
2364	IANA "WebAuthn Attestation Statement Format Identifier" registry
2365	established by [WebAuthn-Registries].
2366	
2367	7.2. Packed Attestation Statement Format
2368	
2369	This is a WebAuthn optimized attestation statement format. It uses a
2370	very compact but still extensible encoding method. It is implementable
2371	by authenticators with limited resources (e.g., secure elements).
2372	
2373	Attestation statement format identifier
2374	packed
2375	
2376	Attestation types supported
2377	All
2378	
2379	Syntax
2380	The syntax of a Packed Attestation statement is defined by the
2381	following CDDL:
2382	
2383	\$attStmtType ::= (
2384	fmt: "packed",
2385	attStmt: packedStmtFormat
2386)
2387	
2388	packedStmtFormat = {
2389	alg: rsaAlgName / eccAlgName,
2390	sig: bytes,
2391	x5c: [attestnCert: bytes, * (caCert: bytes)]
2392	//
2393	{
2394	alg: "ED256" / "ED512",
2395	sig: bytes,
2396	ecdaaKeyId: bytes
2397	}
2398	
2399	The semantics of the fields are as follows:
2400	
2401	alg
2402	A text string containing the name of the algorithm used to
2403	generate the attestation signature. The types rsaAlgName
2404	and eccAlgName are as defined in 5.3.1 Attestation data.
2405	"ED256" and "ED512" refer to algorithms defined in
2406	[FIDOEcdaaAlgorithm].
2407	
2408	sig
2409	A byte string containing the attestation signature.
2410	
2411	x5c

2515	Unregistered attestation statement format identifiers SHOULD use
2516	lowercase reverse domain-name naming, using a domain name registered by
2517	the developer, in order to assure uniqueness of the identifier. All
2518	attestation statement format identifiers MUST be a maximum of 32 octets
2519	in length and MUST consist only of printable USASCII characters,
2520	excluding backslash and doublequote, i.e., VCHAR as defined in
2521	[RFC5234] but without %x22 and %x5c.
2522	
2523	Note: This means attestation statement format identifiers based on
2524	domain names MUST incorporate only LDH Labels [RFC5890].
2525	
2526	Implementations MUST match WebAuthn attestation statement format
2527	identifiers in a case-sensitive fashion.
2528	
2529	Attestation statement formats that may exist in multiple versions
2530	SHOULD include a version in their identifier. In effect, different
2531	versions are thus treated as different formats, e.g., packed2 as a new
2532	version of the packed attestation statement format.
2533	
2534	The following sections present a set of currently-defined and
2535	registered attestation statement formats and their identifiers. The
2536	up-to-date list of registered WebAuthn Extensions is maintained in the
2537	IANA "WebAuthn Attestation Statement Format Identifier" registry
2538	established by [WebAuthn-Registries].
2539	
2540	7.2. Packed Attestation Statement Format
2541	
2542	This is a WebAuthn optimized attestation statement format. It uses a
2543	very compact but still extensible encoding method. It is implementable
2544	by authenticators with limited resources (e.g., secure elements).
2545	
2546	Attestation statement format identifier
2547	packed
2548	
2549	Attestation types supported
2550	All
2551	
2552	Syntax
2553	The syntax of a Packed Attestation statement is defined by the
2554	following CDDL:
2555	
2556	\$attStmtType ::= (
2557	fmt: "packed",
2558	attStmt: packedStmtFormat
2559)
2560	
2561	packedStmtFormat = {
2562	alg: rsaAlgName / eccAlgName,
2563	sig: bytes,
2564	x5c: [attestnCert: bytes, * (caCert: bytes)]
2565	//
2566	{
2567	alg: "ED256" / "ED512",
2568	sig: bytes,
2569	ecdaaKeyId: bytes
2570	}
2571	
2572	The semantics of the fields are as follows:
2573	
2574	alg
2575	A text string containing the name of the algorithm used to
2576	generate the attestation signature. The types rsaAlgName
2577	and eccAlgName are as defined in 5.3.1 Attestation data.
2578	"ED256" and "ED512" refer to algorithms defined in
2579	[FIDOEcdaaAlgorithm].
2580	
2581	sig
2582	A byte string containing the attestation signature.
2583	
2584	x5c

2412 The elements of this array contain the attestation
2413 certificate and its certificate chain, each encoded in
2414 X.509 format. The attestation certificate must be the
2415 first element in the array.
2416
2417 ecdaaKeyld
2418 The identifier of the ECDAA-Issuer public key. This is the
2419 BigIntegerToB encoding of the component "c" of the
2420 ECDAA-Issuer public key as defined section 3.3, step 3.5
2421 in [FIDOEcdaaAlgorithm].
2422
2423 Signing procedure
2424 The signing procedure for this attestation statement format is
2425 similar to the procedure for generating assertion signatures.
2426
2427 Let authenticatorData denote the authenticator data for the
2428 attestation, and let clientDataHash denote the hash of the
2429 serialized client data.
2430
2431 If Basic or Privacy CA attestation is in use, the authenticator
2432 produces the sig by concatenating authenticatorData and
2433 clientDataHash, and signing the result using an attestation
2434 private key selected through an authenticator-specific
2435 mechanism. It sets x5c to the certificate chain of the
2436 attestation public key and alg to the algorithm of the
2437 attestation private key.
2438
2439 If ECDAA is in use, the authenticator produces sig by
2440 concatenating authenticatorData and clientDataHash, and signing
2441 the result using ECDAA-Sign (see section 3.5 of
2442 [FIDOEcdaaAlgorithm]) with a ECDAA-Issuer public key selected
2443 through an authenticator-specific mechanism (see
2444 [FIDOEcdaaAlgorithm]). It sets alg to the algorithm of the
2445 ECDAA-Issuer public key and ecdaaKeyld to the identifier of the
2446 ECDAA-Issuer public key (see above).
2447
2448 If self attestation is in use, the authenticator produces sig by
2449 concatenating authenticatorData and clientDataHash, and signing
2450 the result using the credential private key. It sets alg to the
2451 algorithm of the credential private key, and omits the other
2452 fields.
2453
2454 Verification procedure
2455 Verify that the given attestation statement is valid CBOR
2456 conforming to the syntax defined above.
2457
2458 Let authenticatorData denote the authenticator data claimed to
2459 have been used for the attestation, and let clientDataHash
2460 denote the hash of the serialized client data.
2461
2462 If x5c is present, this indicates that the attestation type is
2463 not ECDAA. In this case:
2464
2465 + Verify that sig is a valid signature over the concatenation of
2466 authenticatorData and clientDataHash using the attestation
2467 public key in x5c with the algorithm specified in alg.
2468 + Verify that x5c meets the requirements in 7.2.1 Packed
2469 attestation statement certificate requirements.
2470 + If x5c contains an extension with OID 1 3 6 1 4 1 45724 1 1 4
2471 (id-fido-gen-ce-aaguid) verify that the value of this
2472 extension matches the AAGUID in authenticatorData.
2473 + If successful, return attestation type Basic and trust path
2474 x5c.
2475
2476 If ecdaaKeyld is present, then the attestation type is ECDAA. In
2477 this case:
2478
2479 + Verify that sig is a valid signature over the concatenation of
2480 authenticatorData and clientDataHash using ECDAA-Verify with
2481 ECDAA-Issuer public key identified by ecdaaKeyld (see

2585 The elements of this array contain the attestation
2586 certificate and its certificate chain, each encoded in
2587 X.509 format. The attestation certificate must be the
2588 first element in the array.
2589
2590 ecdaaKeyld
2591 The identifier of the ECDAA-Issuer public key. This is the
2592 BigIntegerToB encoding of the component "c" of the
2593 ECDAA-Issuer public key as defined section 3.3, step 3.5
2594 in [FIDOEcdaaAlgorithm].
2595
2596 Signing procedure
2597 The signing procedure for this attestation statement format is
2598 similar to the procedure for generating assertion signatures.
2599
2600 Let authenticatorData denote the authenticator data for the
2601 attestation, and let clientDataHash denote the hash of the
2602 serialized client data.
2603
2604 If Basic or Privacy CA attestation is in use, the authenticator
2605 produces the sig by concatenating authenticatorData and
2606 clientDataHash, and signing the result using an attestation
2607 private key selected through an authenticator-specific
2608 mechanism. It sets x5c to the certificate chain of the
2609 attestation public key and alg to the algorithm of the
2610 attestation private key.
2611
2612 If ECDAA is in use, the authenticator produces sig by
2613 concatenating authenticatorData and clientDataHash, and signing
2614 the result using ECDAA-Sign (see section 3.5 of
2615 [FIDOEcdaaAlgorithm]) with a ECDAA-Issuer public key selected
2616 through an authenticator-specific mechanism (see
2617 [FIDOEcdaaAlgorithm]). It sets alg to the algorithm of the
2618 ECDAA-Issuer public key and ecdaaKeyld to the identifier of the
2619 ECDAA-Issuer public key (see above).
2620
2621 If self attestation is in use, the authenticator produces sig by
2622 concatenating authenticatorData and clientDataHash, and signing
2623 the result using the credential private key. It sets alg to the
2624 algorithm of the credential private key, and omits the other
2625 fields.
2626
2627 Verification procedure
2628 Verify that the given attestation statement is valid CBOR
2629 conforming to the syntax defined above.
2630
2631 Let authenticatorData denote the authenticator data claimed to
2632 have been used for the attestation, and let clientDataHash
2633 denote the hash of the serialized client data.
2634
2635 If x5c is present, this indicates that the attestation type is
2636 not ECDAA. In this case:
2637
2638 + Verify that sig is a valid signature over the concatenation of
2639 authenticatorData and clientDataHash using the attestation
2640 public key in x5c with the algorithm specified in alg.
2641 + Verify that x5c meets the requirements in 7.2.1 Packed
2642 attestation statement certificate requirements.
2643 + If x5c contains an extension with OID 1 3 6 1 4 1 45724 1 1 4
2644 (id-fido-gen-ce-aaguid) verify that the value of this
2645 extension matches the AAGUID in authenticatorData.
2646 + If successful, return attestation type Basic and trust path
2647 x5c.
2648
2649 If ecdaaKeyld is present, then the attestation type is ECDAA. In
2650 this case:
2651
2652 + Verify that sig is a valid signature over the concatenation of
2653 authenticatorData and clientDataHash using ECDAA-Verify with
2654 ECDAA-Issuer public key identified by ecdaaKeyld (see

```
2482 [FIDOEcdaaAlgorithm]).
2483 + If successful, return attestation type ECDA and trust path
2484 ecdaaKeyld.
2485
2486 If neither x5c nor ecdaaKeyld is present, self attestation is in
2487 use.
2488
2489 + Validate that alg matches the algorithm of the credential
2490 private key in authenticatorData.
2491 + Verify that sig is a valid signature over the concatenation of
2492 authenticatorData and clientDataHash using the credential
2493 public key with alg.
2494 + If successful, return attestation type Self and empty trust
2495 path.
2496
2497 7.2.1. Packed attestation statement certificate requirements
2498
2499 The attestation certificate MUST have the following fields/extensions:
2500 * Version must be set to 3.
2501 * Subject field MUST be set to:
2502
2503 Subject-C
2504 Country where the Authenticator vendor is incorporated
2505
2506 Subject-O
2507 Legal name of the Authenticator vendor
2508
2509 Subject-OU
2510 Authenticator Attestation
2511
2512 Subject-CN
2513 No stipulation.
2514
2515 * If the related attestation root certificate is used for multiple
2516 authenticator models, the Extension OID 1 3 6 1 4 1 45724 1 1 4
2517 (id-fido-gen-ce-aaguid) MUST be present, containing the AAGUID as
2518 value.
2519 * The Basic Constraints extension MUST have the CA component set to
2520 false
2521 * An Authority Information Access (AIA) extension with entry
2522 id-ad-ocsp and a CRL Distribution Point extension [RFC5280] are
2523 both optional as the status of many attestation certificates is
2524 available through authenticator metadata services. See, for
2525 example, the FIDO Metadata Service [FIDOMetadataService].
2526
2527 7.3. TPM Attestation Statement Format
2528
2529 This attestation statement format is generally used by authenticators
2530 that use a Trusted Platform Module as their cryptographic engine.
2531
2532 Attestation statement format identifier
2533 tpm
2534
2535 Attestation types supported
2536 Privacy CA, ECDA
2537
2538 Syntax
2539 The syntax of a TPM Attestation statement is as follows:
2540
2541 $$attStmtType // = (
2542     fmt: "tpm",
2543     attStmt: tpmStmtFormat
2544 )
2545
2546 tpmStmtFormat = {
2547     ver: "2.0",
2548     (
2549         alg: rsaAlgName / eccAlgName,
2550         x5c: [ aikCert: bytes, * (caCert: bytes) ]
2551     ) //
```

```
2655 [FIDOEcdaaAlgorithm]).
2656 + If successful, return attestation type ECDA and trust path
2657 ecdaaKeyld.
2658
2659 If neither x5c nor ecdaaKeyld is present, self attestation is in
2660 use.
2661
2662 + Validate that alg matches the algorithm of the credential
2663 private key in authenticatorData.
2664 + Verify that sig is a valid signature over the concatenation of
2665 authenticatorData and clientDataHash using the credential
2666 public key with alg.
2667 + If successful, return attestation type Self and empty trust
2668 path.
2669
2670 7.2.1. Packed attestation statement certificate requirements
2671
2672 The attestation certificate MUST have the following fields/extensions:
2673 * Version must be set to 3.
2674 * Subject field MUST be set to:
2675
2676 Subject-C
2677 Country where the Authenticator vendor is incorporated
2678
2679 Subject-O
2680 Legal name of the Authenticator vendor
2681
2682 Subject-OU
2683 Authenticator Attestation
2684
2685 Subject-CN
2686 No stipulation.
2687
2688 * If the related attestation root certificate is used for multiple
2689 authenticator models, the Extension OID 1 3 6 1 4 1 45724 1 1 4
2690 (id-fido-gen-ce-aaguid) MUST be present, containing the AAGUID as
2691 value.
2692 * The Basic Constraints extension MUST have the CA component set to
2693 false
2694 * An Authority Information Access (AIA) extension with entry
2695 id-ad-ocsp and a CRL Distribution Point extension [RFC5280] are
2696 both optional as the status of many attestation certificates is
2697 available through authenticator metadata services. See, for
2698 example, the FIDO Metadata Service [FIDOMetadataService].
2699
2700 7.3. TPM Attestation Statement Format
2701
2702 This attestation statement format is generally used by authenticators
2703 that use a Trusted Platform Module as their cryptographic engine.
2704
2705 Attestation statement format identifier
2706 tpm
2707
2708 Attestation types supported
2709 Privacy CA, ECDA
2710
2711 Syntax
2712 The syntax of a TPM Attestation statement is as follows:
2713
2714 $$attStmtType // = (
2715     fmt: "tpm",
2716     attStmt: tpmStmtFormat
2717 )
2718
2719 tpmStmtFormat = {
2720     ver: "2.0",
2721     (
2722         alg: rsaAlgName / eccAlgName,
2723         x5c: [ aikCert: bytes, * (caCert: bytes) ]
2724     ) //
```

```
2552 (
2553     alg: "ED256" / "ED512",
2554     ecdaaKeyId: bytes
2555 ),
2556 sig: bytes,
2557 certInfo: bytes,
2558 pubArea: bytes
2559 }
2560
2561 The semantics of the above fields are as follows:
2562
2563 ver
2564     The version of the TPM specification to which the
2565     signature conforms.
2566
2567 alg
2568     The name of the algorithm used to generate the attestation
2569     signature. The types rsaAlgName and eccAlgName are as
2570     defined in 5.3.1 Attestation data. The types "ED256" and
2571     "ED512" refer to the algorithms specified in
2572     [FIDOEcdaaAlgorithm].
2573
2574 x5c
2575     The AIK certificate used for the attestation and its
2576     certificate chain, in X.509 encoding.
2577
2578 ecdaaKeyId
2579     The identifier of the ECDAAs-Issuer public key. This is the
2580     BigIntegerToB encoding of the component "c" as defined
2581     section 3.3, step 3.5 in [FIDOEcdaaAlgorithm].
2582
2583 sig
2584     The attestation signature, in the form of a TPMT_SIGNATURE
2585     structure as specified in [TPMv2-Part2] section 11.3.4.
2586
2587 certInfo
2588     The TPMS_ATTEST structure over which the above signature
2589     was computed, as specified in [TPMv2-Part2] section
2590     10.12.8.
2591
2592 pubArea
2593     The TPMT_PUBLIC structure (see [TPMv2-Part2] section
2594     12.2.4) used by the TPM to represent the credential public
2595     key.
2596
2597 Signing procedure
2598     Let authenticatorData denote the authenticator data for the
2599     attestation, and let clientDataHash denote the hash of the
2600     serialized client data.
2601
2602     Concatenate authenticatorData and clientDataHash to form
2603     attToBeSigned.
2604
2605     Generate a signature using the procedure specified in
2606     [TPMv2-Part3] Section 18.2, using the attestation private key
2607     and setting the qualifyingData parameter to attToBeSigned.
2608
2609     Set the pubArea field to the public area of the credential
2610     public key, the certInfo field to the output parameter of the
2611     same name, and the sig field to the signature obtained from the
2612     above procedure.
2613
2614 Verification procedure
2615     Verify that the given attestation statement is valid CBOR
2616     conforming to the syntax defined above.
2617
2618     Let authenticatorData denote the authenticator data claimed to
2619     have been used for the attestation, and let clientDataHash
2620     denote the hash of the serialized client data.
```

```
2725 (
2726     alg: "ED256" / "ED512",
2727     ecdaaKeyId: bytes
2728 ),
2729 sig: bytes,
2730 certInfo: bytes,
2731 pubArea: bytes
2732 }
2733
2734 The semantics of the above fields are as follows:
2735
2736 ver
2737     The version of the TPM specification to which the
2738     signature conforms.
2739
2740 alg
2741     The name of the algorithm used to generate the attestation
2742     signature. The types rsaAlgName and eccAlgName are as
2743     defined in 5.3.1 Attestation data. The types "ED256" and
2744     "ED512" refer to the algorithms specified in
2745     [FIDOEcdaaAlgorithm].
2746
2747 x5c
2748     The AIK certificate used for the attestation and its
2749     certificate chain, in X.509 encoding.
2750
2751 ecdaaKeyId
2752     The identifier of the ECDAAs-Issuer public key. This is the
2753     BigIntegerToB encoding of the component "c" as defined
2754     section 3.3, step 3.5 in [FIDOEcdaaAlgorithm].
2755
2756 sig
2757     The attestation signature, in the form of a TPMT_SIGNATURE
2758     structure as specified in [TPMv2-Part2] section 11.3.4.
2759
2760 certInfo
2761     The TPMS_ATTEST structure over which the above signature
2762     was computed, as specified in [TPMv2-Part2] section
2763     10.12.8.
2764
2765 pubArea
2766     The TPMT_PUBLIC structure (see [TPMv2-Part2] section
2767     12.2.4) used by the TPM to represent the credential public
2768     key.
2769
2770 Signing procedure
2771     Let authenticatorData denote the authenticator data for the
2772     attestation, and let clientDataHash denote the hash of the
2773     serialized client data.
2774
2775     Concatenate authenticatorData and clientDataHash to form
2776     attToBeSigned.
2777
2778     Generate a signature using the procedure specified in
2779     [TPMv2-Part3] Section 18.2, using the attestation private key
2780     and setting the qualifyingData parameter to attToBeSigned.
2781
2782     Set the pubArea field to the public area of the credential
2783     public key, the certInfo field to the output parameter of the
2784     same name, and the sig field to the signature obtained from the
2785     above procedure.
2786
2787 Verification procedure
2788     Verify that the given attestation statement is valid CBOR
2789     conforming to the syntax defined above.
2790
2791     Let authenticatorData denote the authenticator data claimed to
2792     have been used for the attestation, and let clientDataHash
2793     denote the hash of the serialized client data.
```


2622 Verify that the public key specified by the parameters and
2623 unique fields of pubArea is identical to the public key
2624 contained in the attestation data inside authenticatorData.
2625
2626 Concatenate authenticatorData and clientDataHash to form
2627 attToBeSigned.
2628
2629 Validate that certInfo is valid:
2630
2631 + Verify that magic is set to TPM_GENERATED_VALUE.
2632 + Verify that type is set to TPM_ST_ATTEST_CERTIFY.
2633 + Verify that extraData is set to attToBeSigned.
2634 + Verify that attested contains a TPMS_CERTIFY_INFO structure,
2635 whose name field contains a valid Name for pubArea, as
2636 computed using the algorithm in the nameAlg field of pubArea
2637 using the procedure specified in [TPMv2-Part1] section 16.
2638
2639 If x5c is present, this indicates that the attestation type is
2640 not ECDSA. In this case:
2641
2642 + Verify the sig is a valid signature over certInfo using the
2643 attestation public key in x5c with the algorithm specified in
2644 alg.
2645 + Verify that x5c meets the requirements in 7.3.1 TPM
2646 attestation statement certificate requirements.
2647 + If x5c contains an extension with OID 1 3 6 1 4 1 45724 1 1 4
2648 (id-fido-gen-ce-aaguid) verify that the value of this
2649 extension matches the AAGUID in authenticatorData.
2650 + If successful, return attestation type Privacy CA and trust
2651 path x5c.
2652
2653 If ecdaaKeyId is present, then the attestation type is ECDSA.
2654
2655 + Perform ECDSA-Verify on sig to verify that it is a valid
2656 signature over certInfo (see [FIDOEcdaaAlgorithm]).
2657 + If successful, return attestation type ECDSA and the
2658 identifier of the ECDSA-Issuer public key ecdaaKeyId.
2659
2660 7.3.1. TPM attestation statement certificate requirements
2661
2662 TPM attestation certificate MUST have the following fields/extensions:
2663 * Version must be set to 3.
2664 * Subject field MUST be set to empty.
2665 * The Subject Alternative Name extension must be set as defined in
2666 [TPMv2-EK-Profile] section 3.2.9.
2667 * The Extended Key Usage extension MUST contain the
2668 "joint-iso-itu-t(2) internationalorganizations(23) 133 tcg-kp(8)
2669 tcg-kp-AIKCertificate(3)" OID.
2670 * The Basic Constraints extension MUST have the CA component set to
2671 false.
2672 * An Authority Information Access (AIA) extension with entry
2673 id-ad-ocsp and a CRL Distribution Point extension [RFC5280] are
2674 both optional as the status of many attestation certificates is
2675 available through metadata services. See, for example, the FIDO
2676 Metadata Service [FIDOMetadataService].
2677
2678 7.4. Android Key Attestation Statement Format
2679
2680 When the authenticator in question is a platform-provided Authenticator
2681 on the Android "N" or later platform, the attestation statement is
2682 based on the Android key attestation. In these cases, the attestation
2683 statement is produced by a component running in a secure operating
2684 environment, but the authenticator data for the attestation is produced
2685 outside this environment. The Relying Party is expected to check that
2686 the authenticator data claimed to have been used for the attestation is
2687 consistent with the fields of the attestation certificate's extension
2688 data.
2689
2690 Attestation statement format identifier
2691 android-key

2795 Verify that the public key specified by the parameters and
2796 unique fields of pubArea is identical to the public key
2797 contained in the attestation data inside authenticatorData.
2798
2799 Concatenate authenticatorData and clientDataHash to form
2800 attToBeSigned.
2801
2802 Validate that certInfo is valid:
2803
2804 + Verify that magic is set to TPM_GENERATED_VALUE.
2805 + Verify that type is set to TPM_ST_ATTEST_CERTIFY.
2806 + Verify that extraData is set to attToBeSigned.
2807 + Verify that attested contains a TPMS_CERTIFY_INFO structure,
2808 whose name field contains a valid Name for pubArea, as
2809 computed using the algorithm in the nameAlg field of pubArea
2810 using the procedure specified in [TPMv2-Part1] section 16.
2811
2812 If x5c is present, this indicates that the attestation type is
2813 not ECDSA. In this case:
2814
2815 + Verify the sig is a valid signature over certInfo using the
2816 attestation public key in x5c with the algorithm specified in
2817 alg.
2818 + Verify that x5c meets the requirements in 7.3.1 TPM
2819 attestation statement certificate requirements.
2820 + If x5c contains an extension with OID 1 3 6 1 4 1 45724 1 1 4
2821 (id-fido-gen-ce-aaguid) verify that the value of this
2822 extension matches the AAGUID in authenticatorData.
2823 + If successful, return attestation type Privacy CA and trust
2824 path x5c.
2825
2826 If ecdaaKeyId is present, then the attestation type is ECDSA.
2827
2828 + Perform ECDSA-Verify on sig to verify that it is a valid
2829 signature over certInfo (see [FIDOEcdaaAlgorithm]).
2830 + If successful, return attestation type ECDSA and the
2831 identifier of the ECDSA-Issuer public key ecdaaKeyId.
2832
2833 7.3.1. TPM attestation statement certificate requirements
2834
2835 TPM attestation certificate MUST have the following fields/extensions:
2836 * Version must be set to 3.
2837 * Subject field MUST be set to empty.
2838 * The Subject Alternative Name extension must be set as defined in
2839 [TPMv2-EK-Profile] section 3.2.9.
2840 * The Extended Key Usage extension MUST contain the
2841 "joint-iso-itu-t(2) internationalorganizations(23) 133 tcg-kp(8)
2842 tcg-kp-AIKCertificate(3)" OID.
2843 * The Basic Constraints extension MUST have the CA component set to
2844 false.
2845 * An Authority Information Access (AIA) extension with entry
2846 id-ad-ocsp and a CRL Distribution Point extension [RFC5280] are
2847 both optional as the status of many attestation certificates is
2848 available through metadata services. See, for example, the FIDO
2849 Metadata Service [FIDOMetadataService].
2850
2851 7.4. Android Key Attestation Statement Format
2852
2853 When the authenticator in question is a platform-provided Authenticator
2854 on the Android "N" or later platform, the attestation statement is
2855 based on the Android key attestation. In these cases, the attestation
2856 statement is produced by a component running in a secure operating
2857 environment, but the authenticator data for the attestation is produced
2858 outside this environment. The Relying Party is expected to check that
2859 the authenticator data claimed to have been used for the attestation is
2860 consistent with the fields of the attestation certificate's extension
2861 data.
2862
2863 Attestation statement format identifier
2864 android-key

2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761

Attestation types supported
Basic

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

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

androidStmtFormat = bytes

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

Concatenate authenticatorData and clientDataHash to form attToBeSigned.

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

Verification procedure
Verification is performed as follows:

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

7.5. Android SafetyNet Attestation Statement Format

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

Attestation statement format identifier
android-safetynet

Attestation types supported
Basic

2865
2866
2867
2868
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934

Attestation types supported
Basic

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

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

androidStmtFormat = bytes

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

Concatenate authenticatorData and clientDataHash to form attToBeSigned.

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

Verification procedure
Verification is performed as follows:

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

7.5. Android SafetyNet Attestation Statement Format

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

Attestation statement format identifier
android-safetynet

Attestation types supported
Basic

2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2830
2831

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

```
$$attStmtType //= (  
    fmt: "android-safetynet",  
    attStmt: safetynetStmtFormat  
)  
  
safetynetStmtFormat = {  
    ver: text,  
    response: bytes  
}
```

The semantics of the above fields are as follows:

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

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

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

Concatenate authenticatorData and clientDataHash to form attToBeSigned.

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

Verification procedure
Verification is performed as follows:

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

7.6. FIDO U2F Attestation Statement Format

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

Attestation statement format identifier
fido-u2f

Attestation types supported
Basic

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

2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
3000
3001
3002
3003
3004

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

```
$$attStmtType //= (  
    fmt: "android-safetynet",  
    attStmt: safetynetStmtFormat  
)  
  
safetynetStmtFormat = {  
    ver: text,  
    response: bytes  
}
```

The semantics of the above fields are as follows:

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

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

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

Concatenate authenticatorData and clientDataHash to form attToBeSigned.

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

Verification procedure
Verification is performed as follows:

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

7.6. FIDO U2F Attestation Statement Format

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

Attestation statement format identifier
fido-u2f

Attestation types supported
Basic, self attestation

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


```
2832  $$attStmtType //=(
2833      fmt: "fido-u2f",
2834      attStmt: u2fStmtFormat
2835  )
2836
2837  u2fStmtFormat = {
2838      x5c: [ attestnCert: bytes, * (caCert: bytes) ],
2839      sig: bytes
2840  }
2841
2842  The semantics of the above fields are as follows:
2843
2844  x5c
2845      The elements of this array contain the attestation
2846      certificate and its certificate chain, each encoded in
2847      X.509 format. The attestation certificate must be the
2848      first element in the array.
2849
2850  sig
2851      The attestation signature.
2852
2853  Signing procedure
2854  If the credential public key of the given credential is not of
2855  algorithm "ES256", stop and return an error.
2856
2857  Let authenticatorData denote the authenticator data for the
2858  attestation, and let clientDataHash denote the hash of the
2859  serialized client data.
2860
2861  If clientDataHash is 256 bits long, set tbsHash to this value.
2862  Otherwise set tbsHash to the SHA-256 hash of clientDataHash.
2863
2864  Generate a signature as specified in [FIDO-U2F-Message-Formats]
2865  section 4.3, with the application parameter set to the SHA-256
2866  hash of the RP ID associated with the given credential, the
2867  challenge parameter set to tbsHash, and the key handle parameter
2868  set to the credential ID of the given credential. Set this as
2869  sig and set the attestation certificate of the attestation
2870  public key as x5c.
2871
2872  Verification procedure
2873  Verification is performed as follows:
2874
2875  + Verify that the given attestation statement is valid CBOR
2876  conforming to the syntax defined above.
2877  + If x5c is not a certificate for an ECDSA public key over the
2878  P-256 curve, stop verification and return an error.
2879  + Let authenticatorData denote the authenticator data claimed to
2880  have been used for the attestation, and let clientDataHash
2881  denote the hash of the serialized client data.
2882  + If clientDataHash is 256 bits long, set tbsHash to this value.
2883  Otherwise set tbsHash to the SHA-256 hash of clientDataHash.
2884  + From authenticatorData, extract the claimed RP ID hash, the
2885  claimed credential ID and the claimed credential public key.
2886  + Generate the claimed to-be-signed data as specified in
2887  [FIDO-U2F-Message-Formats] section 4.3, with the application
2888  parameter set to the claimed RP ID hash, the challenge
2889  parameter set to tbsHash, the key handle parameter set to the
2890  claimed credential ID of the given credential, and the user
2891  public key parameter set to the claimed credential public key.
2892  + Verify that the sig is a valid ECDSA P-256 signature over the
2893  to-be-signed data constructed above.
2894  + If successful, return attestation type Basic with the trust
2895  path set to x5c.
2896
2897  8. WebAuthn Extensions
2898
2899  The mechanism for generating public key credentials, as well as
2900  requesting and generating Authentication assertions, as defined in 4
2901  Web Authentication API, can be extended to suit particular use cases.
```

```
3005  $$attStmtType //=(
3006      fmt: "fido-u2f",
3007      attStmt: u2fStmtFormat
3008  )
3009
3010  u2fStmtFormat = {
3011      x5c: [ attestnCert: bytes, * (caCert: bytes) ],
3012      sig: bytes
3013  }
3014
3015  The semantics of the above fields are as follows:
3016
3017  x5c
3018      The elements of this array contain the attestation
3019      certificate and its certificate chain, each encoded in
3020      X.509 format. The attestation certificate must be the
3021      first element in the array.
3022
3023  sig
3024      The attestation signature.
3025
3026  Signing procedure
3027  If the credential public key of the given credential is not of
3028  algorithm -7 ("ES256"), stop and return an error.
3029
3030  Let authenticatorData denote the authenticator data for the
3031  attestation, and let clientDataHash denote the hash of the
3032  serialized client data.
3033
3034  If clientDataHash is 256 bits long, set tbsHash to this value.
3035  Otherwise set tbsHash to the SHA-256 hash of clientDataHash.
3036
3037  Generate a signature as specified in [FIDO-U2F-Message-Formats]
3038  section 4.3, with the application parameter set to the SHA-256
3039  hash of the RP ID associated with the given credential, the
3040  challenge parameter set to tbsHash, and the key handle parameter
3041  set to the credential ID of the given credential. Set this as
3042  sig and set the attestation certificate of the attestation
3043  public key as x5c.
3044
3045  Verification procedure
3046  Verification is performed as follows:
3047
3048  + Verify that the given attestation statement is valid CBOR
3049  conforming to the syntax defined above.
3050  + If x5c is not a certificate for an ECDSA public key over the
3051  P-256 curve, stop verification and return an error.
3052  + Let authenticatorData denote the authenticator data claimed to
3053  have been used for the attestation, and let clientDataHash
3054  denote the hash of the serialized client data.
3055  + If clientDataHash is 256 bits long, set tbsHash to this value.
3056  Otherwise set tbsHash to the SHA-256 hash of clientDataHash.
3057  + From authenticatorData, extract the claimed RP ID hash, the
3058  claimed credential ID and the claimed credential public key.
3059  + Generate the claimed to-be-signed data as specified in
3060  [FIDO-U2F-Message-Formats] section 4.3, with the application
3061  parameter set to the claimed RP ID hash, the challenge
3062  parameter set to tbsHash, the key handle parameter set to the
3063  claimed credential ID of the given credential, and the user
3064  public key parameter set to the claimed credential public key.
3065  + Verify that the sig is a valid ECDSA P-256 signature over the
3066  to-be-signed data constructed above.
3067  + If successful, return attestation type Basic with the trust
3068  path set to x5c.
3069
3070  8. WebAuthn Extensions
3071
3072  The mechanism for generating public key credentials, as well as
3073  requesting and generating Authentication assertions, as defined in 4
3074  Web Authentication API, can be extended to suit particular use cases.
```

Each case is addressed by defining a registration extension and/or an authentication extension.

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

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

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

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

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

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

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

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

Each case is addressed by defining a registration extension and/or an authentication extension.

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

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

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

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

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

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

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

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

clients can choose to produce a client extension output value for an extension that it does not understand by encoding the authenticator extension output value into JSON, provided that the CBOR output uses only types present in JSON.

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

8.1. Extension Identifiers

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

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

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

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

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

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

8.2. Defining extensions

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

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

8.3. Extending request parameters

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

clients can choose to produce a client extension output value for an extension that it does not understand by encoding the authenticator extension output value into JSON, provided that the CBOR output uses only types present in JSON.

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

8.1. Extension Identifiers

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

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

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

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

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

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

8.2. Defining extensions

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

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

8.3. Extending request parameters

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

these calls.

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

```
var assertionPromise = navigator.credentials.get({
  publicKey: {
    challenge: "...",
    extensions: {
      "webauthnExample_foobar": 42
    }
  }
});
```

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

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

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

8.4. Client extension processing

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

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

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

8.5. Authenticator extension processing

As specified in 5.1 Authenticator data, the CBOR authenticator extension input value of each processed authenticator extension is included in the extensions data part of the authenticator data. This part is a CBOR map, with CBOR extension identifier values as keys, and the CBOR authenticator extension input value of each extension as the value.

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

these calls.

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

```
var assertionPromise = navigator.credentials.get({
  publicKey: {
    challenge: "...",
    extensions: {
      "webauthnExample_foobar": 42
    }
  }
});
```

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

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

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

8.4. Client extension processing

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

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

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

8.5. Authenticator extension processing

As specified in 5.1 Authenticator data, the CBOR authenticator extension input value of each processed authenticator extension is included in the extensions data part of the authenticator data. This part is a CBOR map, with CBOR extension identifier values as keys, and the CBOR authenticator extension input value of each extension as the value.

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

3112 The authenticator extension processing rules are used create the
3113 authenticator extension output from the authenticator extension input,
3114 and possibly also other inputs, for each extension.
3115
3116 8.6. Example Extension
3117
3118 This section is not normative.
3119
3120 To illustrate the requirements above, consider a hypothetical
3121 registration extension and authentication extension "Geo". This
3122 extension, if supported, enables a geolocation location to be returned
3123 from the authenticator or client to the Relying Party.
3124
3125 The extension identifier is chosen as webauthnExample_geo. The client
3126 extension input is the constant value true, since the extension does
3127 not require the Relying Party to pass any particular information to the
3128 client, other than that it requests the use of the extension. The
3129 Relying Party sets this value in its request for an assertion:
3130 var assertionPromise =
3131 navigator.credentials.get({
3132 publicKey: {
3133 challenge: "SGFuFNvbG8gc2hvdCBmaXJzdC4",
3134 allowList: [], /* Empty filter */
3135 extensions: { 'webauthnExample_geo': true }
3136 }
3137 });
3138
3139 The extension also requires the client to set the authenticator
3140 parameter to the fixed value true.
3141
3142 The extension requires the authenticator to specify its geolocation in
3143 the authenticator extension output, if known. The extension e.g.
3144 specifies that the location shall be encoded as a two-element array of
3145 floating point numbers, encoded with CBOR. An authenticator does this
3146 by including it in the authenticator data. As an example, authenticator
3147 data may be as follows (notation taken from [RFC7049]):
3148 81 (hex) -- Flags, ED and UP both set.
3149 20 05 58 1F -- Signature counter
3150 A1 -- CBOR map of one element
3151 73 -- Key 1: CBOR text string of 19 byt
3152 es
3153 77 65 62 61 75 74 68 6E 45 78 61
3154 6D 70 6C 65 5F 67 65 6F -- "webauthnExample_geo" [=UTF-8 enc
3155 oded=] string
3156 82 -- Value 1: CBOR array of two elemen
3157 ts
3158 FA 42 82 1E B3 -- Element 1: Latitude as CBOR encod
3159 ed float
3160 FA C1 5F E3 7F -- Element 2: Longitude as CBOR enco
3161 ded float
3162
3163 The extension defines the client extension output to be the geolocation
3164 information, if known, as a GeoJSON [GeoJSON] point. The client
3165 constructs the following client data:
3166 {
3167 'extensions': {
3168 'webauthnExample_geo': {
3169 'type': 'Point',
3170 'coordinates': [65.059962, -13.993041]
3171 }
3172 }
3173 }
3174
3175 9. Defined Extensions
3176
3177 This section defines the initial set of extensions to be registered in
3178 the IANA "WebAuthn Extension Identifier" registry established by
3179 [WebAuthn-Registries]. These are recommended for implementation by user

3285 The authenticator extension processing rules are used create the
3286 authenticator extension output from the authenticator extension input,
3287 and possibly also other inputs, for each extension.
3288
3289 8.6. Example Extension
3290
3291 This section is not normative.
3292
3293 To illustrate the requirements above, consider a hypothetical
3294 registration extension and authentication extension "Geo". This
3295 extension, if supported, enables a geolocation location to be returned
3296 from the authenticator or client to the Relying Party.
3297
3298 The extension identifier is chosen as webauthnExample_geo. The client
3299 extension input is the constant value true, since the extension does
3300 not require the Relying Party to pass any particular information to the
3301 client, other than that it requests the use of the extension. The
3302 Relying Party sets this value in its request for an assertion:
3303 var assertionPromise =
3304 navigator.credentials.get({
3305 publicKey: {
3306 challenge: "SGFuFNvbG8gc2hvdCBmaXJzdC4",
3307 allowCredentiaals: [], /* Empty filter */
3308 extensions: { 'webauthnExample_geo': true }
3309 }
3310 });
3311
3312 The extension also requires the client to set the authenticator
3313 parameter to the fixed value true.
3314
3315 The extension requires the authenticator to specify its geolocation in
3316 the authenticator extension output, if known. The extension e.g.
3317 specifies that the location shall be encoded as a two-element array of
3318 floating point numbers, encoded with CBOR. An authenticator does this
3319 by including it in the authenticator data. As an example, authenticator
3320 data may be as follows (notation taken from [RFC7049]):
3321 81 (hex) -- Flags, ED and UP both set.
3322 20 05 58 1F -- Signature counter
3323 A1 -- CBOR map of one element
3324 73 -- Key 1: CBOR text string of 19 byt
3325 es
3326 77 65 62 61 75 74 68 6E 45 78 61
3327 6D 70 6C 65 5F 67 65 6F -- "webauthnExample_geo" [=UTF-8 enc
3328 oded=] string
3329 82 -- Value 1: CBOR array of two elemen
3330 ts
3331 FA 42 82 1E B3 -- Element 1: Latitude as CBOR encod
3332 ed float
3333 FA C1 5F E3 7F -- Element 2: Longitude as CBOR enco
3334 ded float
3335
3336 The extension defines the client extension output to be the geolocation
3337 information, if known, as a GeoJSON [GeoJSON] point. The client
3338 constructs the following client data:
3339 {
3340 'extensions': {
3341 'webauthnExample_geo': {
3342 'type': 'Point',
3343 'coordinates': [65.059962, -13.993041]
3344 }
3345 }
3346 }
3347
3348 9. Defined Extensions
3349
3350 This section defines the initial set of extensions to be registered in
3351 the IANA "WebAuthn Extension Identifier" registry established by
3352 [WebAuthn-Registries]. These are recommended for implementation by user

agents targeting broad interoperability.

9.1. FIDO AppId Extension (appid)

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

Extension identifier
 appid

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

Client extension processing
 If rpId is present, reject promise with a DOMException whose name is "NotAllowedError", and terminate this algorithm. Replace the calculation of rpId in Step 3 of 4.1.4 Use an existing credential - **PublicKeyCredential::[[DiscoverFromExternalSource]](options) method with the following procedure: The client uses the value of appid to perform the AppId validation procedure (as defined by [FIDO-APPID]).** If valid, the value of rpId for all client processing should be replaced by the value of appid.

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

Authenticator extension input
 None.

Authenticator extension processing
 None.

Authenticator extension output
 None.

9.2. Simple Transaction Authorization Extension (txAuthSimple)

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

Extension identifier
 txAuthSimple

Client extension input
 A single JSON string prompt.

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

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

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

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

agents targeting broad interoperability.

9.1. FIDO AppId Extension (appid)

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

Extension identifier
 appid

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

Client extension processing
 If rpId is present, reject promise with a DOMException whose name is "NotAllowedError", and terminate this algorithm. Replace the calculation of rpId in Step 3 of 4.1.4 Use an existing credential **to make an assertion - PublicKeyCredential's [[DiscoverFromExternalSource]](options) method with the following procedure: The client uses the value of appid to perform the AppId validation procedure (as defined by [FIDO-APPID]).** If valid, the value of rpId for all client processing should be replaced by the value of appid.

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

Authenticator extension input
 None.

Authenticator extension processing
 None.

Authenticator extension output
 None.

9.2. Simple Transaction Authorization Extension (txAuthSimple)

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

Extension identifier
 txAuthSimple

Client extension input
 A single JSON string prompt.

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

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

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

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

3252 Authenticator extension output
3253 A single CBOR string, representing the prompt as displayed
3254 (including any eventual line breaks).
3255
3256
3257 9.3. Generic Transaction Authorization Extension (txAuthGeneric)
3258
3259 This registration extension and authentication extension allows images
3260 to be used as transaction authorization prompts as well. This allows
3261 authenticators without a font rendering engine to be used and also
3262 supports a richer visual appearance.
3263
3264 Extension identifier
3265 txAuthGeneric
3266
3267 Client extension input
3268 A CBOR map defined as follows:
3269
3270 txAuthGenericArg = {
3271 contentType: text, ; MIME-Type of the content, e.g.
3272 "image/png"
3273 content: bytes
3274 }
3275
3276 Client extension processing
3277 None, except creating the authenticator extension input from the
3278 client extension input.
3279
3280 Client extension output
3281 Returns the base64url encoding of the authenticator extension
3282 output value as a JSON string
3283
3284 Authenticator extension input
3285 The client extension input encoded as a CBOR map.
3286
3287 Authenticator extension processing
3288 The authenticator MUST display the content to the user before
3289 performing either user verification or test of user presence.
3290 The authenticator may add other information below the content.
3291 No changes are allowed to the content itself, i.e., inside
3292 content boundary box.
3293
3294 Authenticator extension output
3295 The hash value of the content which was displayed. The
3296 authenticator MUST use the same hash algorithm as it uses for
3297 the signature itself.
3298
3299 9.4. Authenticator Selection Extension (authnSel)
3300
3301 This registration extension allows a Relying Party to guide the
3302 selection of the authenticator that will be leveraged when creating the
3303 credential. It is intended primarily for Relying Parties that wish to
3304 tightly control the experience around credential creation.
3305
3306 Extension identifier
3307 authnSel
3308
3309 Client extension input
3310 A sequence of AAGUIDs:
3311
3312 typedef sequence<AAGUID> AuthenticatorSelectionList;
3313
3314 Each AAGUID corresponds to an authenticator model that is
3315 acceptable to the Relying Party for this credential creation.
3316 The list is ordered by decreasing preference.
3317
3318 An AAGUID is defined as an array containing the globally unique
3319 identifier of the authenticator model being sought.
3320
3321 typedef BufferSource AAGUID;

3425 Authenticator extension output
3426 A single CBOR string, representing the prompt as displayed
3427 (including any eventual line breaks).
3428
3429
3430 9.3. Generic Transaction Authorization Extension (txAuthGeneric)
3431
3432 This registration extension and authentication extension allows images
3433 to be used as transaction authorization prompts as well. This allows
3434 authenticators without a font rendering engine to be used and also
3435 supports a richer visual appearance.
3436
3437 Extension identifier
3438 txAuthGeneric
3439
3440 Client extension input
3441 A CBOR map defined as follows:
3442
3443 txAuthGenericArg = {
3444 contentType: text, ; MIME-Type of the content, e.g.
3445 "image/png"
3446 content: bytes
3447 }
3448
3449 Client extension processing
3450 None, except creating the authenticator extension input from the
3451 client extension input.
3452
3453 Client extension output
3454 Returns the base64url encoding of the authenticator extension
3455 output value as a JSON string
3456
3457 Authenticator extension input
3458 The client extension input encoded as a CBOR map.
3459
3460 Authenticator extension processing
3461 The authenticator MUST display the content to the user before
3462 performing either user verification or test of user presence.
3463 The authenticator may add other information below the content.
3464 No changes are allowed to the content itself, i.e., inside
3465 content boundary box.
3466
3467 Authenticator extension output
3468 The hash value of the content which was displayed. The
3469 authenticator MUST use the same hash algorithm as it uses for
3470 the signature itself.
3471
3472 9.4. Authenticator Selection Extension (authnSel)
3473
3474 This registration extension allows a Relying Party to guide the
3475 selection of the authenticator that will be leveraged when creating the
3476 credential. It is intended primarily for Relying Parties that wish to
3477 tightly control the experience around credential creation.
3478
3479 Extension identifier
3480 authnSel
3481
3482 Client extension input
3483 A sequence of AAGUIDs:
3484
3485 typedef sequence<AAGUID> AuthenticatorSelectionList;
3486
3487 Each AAGUID corresponds to an authenticator model that is
3488 acceptable to the Relying Party for this credential creation.
3489 The list is ordered by decreasing preference.
3490
3491 An AAGUID is defined as an array containing the globally unique
3492 identifier of the authenticator model being sought.
3493
3494 typedef BufferSource AAGUID;

3322
3323 Client extension processing
3324 This extension can only be used during create(). If the client
3325 supports the Authenticator Selection Extension, it MUST use the
3326 first available authenticator whose AAGUID is present in the
3327 AuthenticatorSelectionList. If none of the available
3328 authenticators match a provided AAGUID, the client MUST select
3329 an authenticator from among the available authenticators to
3330 generate the credential.
3331
3332 Client extension output
3333 Returns the JSON value true to indicate to the RP that the
3334 extension was acted upon
3335
3336 Authenticator extension input
3337 None.
3338
3339 Authenticator extension processing
3340 None.
3341
3342 Authenticator extension output
3343 None.
3344
3345 9.5. Supported Extensions Extension (exts)
3346
3347 This registration extension enables the Relying Party to determine
3348 which extensions the authenticator supports.
3349
3350 Extension identifier
3351 exts
3352
3353 Client extension input
3354 The Boolean value true to indicate that this extension is
3355 requested by the Relying Party.
3356
3357 Client extension processing
3358 None, except creating the authenticator extension input from the
3359 client extension input.
3360
3361 Client extension output
3362 Returns the list of supported extensions as a JSON array of
3363 extension identifier strings
3364
3365 Authenticator extension input
3366 The Boolean value true, encoded in CBOR (major type 7, value
3367 21).
3368
3369 Authenticator extension processing
3370 The authenticator sets the authenticator extension output to be
3371 a list of extensions that the authenticator supports, as defined
3372 below. This extension can be added to attestation objects.
3373
3374 Authenticator extension output
3375 The SupportedExtensions extension is a list (CBOR array) of
3376 extension identifier (UTF-8 encoded strings).
3377
3378 9.6. User Verification Index Extension (uvi)
3379
3380 This registration extension and authentication extension enables use of
3381 a user verification index.
3382
3383 Extension identifier
3384 uvi
3385
3386 Client extension input
3387 The Boolean value true to indicate that this extension is
3388 requested by the Relying Party.
3389
3390 Client extension processing
3391 None, except creating the authenticator extension input from the

3496
3497 Client extension processing
3498 This extension can only be used during create(). If the client
3499 supports the Authenticator Selection Extension, it MUST use the
3500 first available authenticator whose AAGUID is present in the
3501 AuthenticatorSelectionList. If none of the available
3502 authenticators match a provided AAGUID, the client MUST select
3503 an authenticator from among the available authenticators to
3504 generate the credential.
3505
3506 Client extension output
3507 Returns the JSON value true to indicate to the RP that the
3508 extension was acted upon
3509
3510 Authenticator extension input
3511 None.
3512
3513 Authenticator extension processing
3514 None.
3515
3516 Authenticator extension output
3517 None.
3518
3519 9.5. Supported Extensions Extension (exts)
3520
3521 This registration extension enables the Relying Party to determine
3522 which extensions the authenticator supports.
3523
3524 Extension identifier
3525 exts
3526
3527 Client extension input
3528 The Boolean value true to indicate that this extension is
3529 requested by the Relying Party.
3530
3531 Client extension processing
3532 None, except creating the authenticator extension input from the
3533 client extension input.
3534
3535 Client extension output
3536 Returns the list of supported extensions as a JSON array of
3537 extension identifier strings
3538
3539 Authenticator extension input
3540 The Boolean value true, encoded in CBOR (major type 7, value
3541 21).
3542
3543 Authenticator extension processing
3544 The authenticator sets the authenticator extension output to be
3545 a list of extensions that the authenticator supports, as defined
3546 below. This extension can be added to attestation objects.
3547
3548 Authenticator extension output
3549 The SupportedExtensions extension is a list (CBOR array) of
3550 extension identifier (UTF-8 encoded strings).
3551
3552 9.6. User Verification Index Extension (uvi)
3553
3554 This registration extension and authentication extension enables use of
3555 a user verification index.
3556
3557 Extension identifier
3558 uvi
3559
3560 Client extension input
3561 The Boolean value true to indicate that this extension is
3562 requested by the Relying Party.
3563
3564 Client extension processing
None, except creating the authenticator extension input from the

3392 client extension input.
3393
3394 Client extension output
3395 Returns a JSON string containing the base64url encoding of the
3396 authenticator extension output
3397
3398 Authenticator extension input
3399 The Boolean value true, encoded in CBOR (major type 7, value
3400 21).
3401
3402 Authenticator extension processing
3403 The authenticator sets the authenticator extension output to be
3404 a user verification index indicating the method used by the user
3405 to authorize the operation, as defined below. This extension can
3406 be added to attestation objects and assertions.
3407
3408 Authenticator extension output
3409 The user verification index (UVI) is a value uniquely
3410 identifying a user verification data record. The UVI is encoded
3411 as CBOR byte string (type 0x58). Each UVI value MUST be specific
3412 to the related key (in order to provide unlinkability). It also
3413 must contain sufficient entropy that makes guessing impractical.
3414 UVI values MUST NOT be reused by the Authenticator (for other
3415 biometric data or users).
3416
3417 The UVI data can be used by servers to understand whether an
3418 authentication was authorized by the exact same biometric data
3419 as the initial key generation. This allows the detection and
3420 prevention of "friendly fraud".
3421
3422 As an example, the UVI could be computed as SHA256(KeyID |
3423 SHA256(rawUVI)), where the rawUVI reflects (a) the biometric
3424 reference data, (b) the related OS level user ID and (c) an
3425 identifier which changes whenever a factory reset is performed
3426 for the device, e.g. rawUVI = biometricReferenceData |
3427 OSLevelUserID | FactoryResetCounter.
3428
3429 Servers supporting UVI extensions MUST support a length of up to
3430 32 bytes for the UVI value.
3431
3432 Example for authenticator data containing one UVI extension
3433
3434 ... -- RP ID hash (32 bytes)
3435 81 -- TUP and ED set
3436 00 00 00 01 -- (initial) signature counter
3437 ... -- all public key alg etc.
3438 A1 -- extension: CBOR map of one elemen
3439 t
3440 63 -- Key 1: CBOR text string of 3 byte
3441 s
3442 75 76 69 -- "uvi" [=UTF-8 encoded=] string
3443 58 20 -- Value 1: CBOR byte string with 0x
3444 20 bytes
3445 00 43 B8 E3 BE 27 95 8C -- the UVI value itself
3446 28 D5 74 BF 46 8A 85 CF
3447 46 9A 14 F0 E5 16 69 31
3448 DA 4B CF FF C1 BB 11 32
3449 82
3450
3451 9.7. Location Extension (loc)
3452
3453 The location registration extension and authentication extension
3454 provides the client device's current location to the WebAuthn relying
3455 party.
3456
3457 Extension identifier
3458 loc
3459
3460 Client extension input
3461 The Boolean value true to indicate that this extension is

3565 client extension input.
3566
3567 Client extension output
3568 Returns a JSON string containing the base64url encoding of the
3569 authenticator extension output
3570
3571 Authenticator extension input
3572 The Boolean value true, encoded in CBOR (major type 7, value
3573 21).
3574
3575 Authenticator extension processing
3576 The authenticator sets the authenticator extension output to be
3577 a user verification index indicating the method used by the user
3578 to authorize the operation, as defined below. This extension can
3579 be added to attestation objects and assertions.
3580
3581 Authenticator extension output
3582 The user verification index (UVI) is a value uniquely
3583 identifying a user verification data record. The UVI is encoded
3584 as CBOR byte string (type 0x58). Each UVI value MUST be specific
3585 to the related key (in order to provide unlinkability). It also
3586 must contain sufficient entropy that makes guessing impractical.
3587 UVI values MUST NOT be reused by the Authenticator (for other
3588 biometric data or users).
3589
3590 The UVI data can be used by servers to understand whether an
3591 authentication was authorized by the exact same biometric data
3592 as the initial key generation. This allows the detection and
3593 prevention of "friendly fraud".
3594
3595 As an example, the UVI could be computed as SHA256(KeyID |
3596 SHA256(rawUVI)), where the rawUVI reflects (a) the biometric
3597 reference data, (b) the related OS level user ID and (c) an
3598 identifier which changes whenever a factory reset is performed
3599 for the device, e.g. rawUVI = biometricReferenceData |
3600 OSLevelUserID | FactoryResetCounter.
3601
3602 Servers supporting UVI extensions MUST support a length of up to
3603 32 bytes for the UVI value.
3604
3605 Example for authenticator data containing one UVI extension
3606
3607 ... -- [=RP ID=] hash (32 bytes)
3608 81 -- UP and ED set
3609 00 00 00 01 -- (initial) signature counter
3610 ... -- all public key alg etc.
3611 A1 -- extension: CBOR map of one elemen
3612 t
3613 63 -- Key 1: CBOR text string of 3 byte
3614 s
3615 75 76 69 -- "uvi" [=UTF-8 encoded=] string
3616 58 20 -- Value 1: CBOR byte string with 0x
3617 20 bytes
3618 00 43 B8 E3 BE 27 95 8C -- the UVI value itself
3619 28 D5 74 BF 46 8A 85 CF
3620 46 9A 14 F0 E5 16 69 31
3621 DA 4B CF FF C1 BB 11 32
3622 82
3623
3624 9.7. Location Extension (loc)
3625
3626 The location registration extension and authentication extension
3627 provides the client device's current location to the WebAuthn Relying
3628 Party.
3629
3630 Extension identifier
3631 loc
3632
3633 Client extension input
3634 The Boolean value true to indicate that this extension is

3462 requested by the Relying Party.
3463
3464 Client extension processing
3465 None, except creating the authenticator extension input from the
3466 client extension input.
3467
3468 Client extension output
3469 Returns a JSON object that encodes the location information in
3470 the authenticator extension output as a Coordinates value, as
3471 defined by The W3C Geolocation API Specification.
3472
3473 Authenticator extension input
3474 The Boolean value true, encoded in CBOR (major type 7, value
3475 21).
3476
3477 Authenticator extension processing
3478 If the authenticator does not support the extension, then the
3479 authenticator MUST ignore the extension request. If the
3480 authenticator accepts the extension, then the authenticator
3481 SHOULD only add this extension data to a packed attestation or
3482 assertion.
3483
3484 Authenticator extension output
3485 If the authenticator accepts the extension request, then
3486 authenticator extension output SHOULD provide location data in
3487 the form of a CBOR-encoded map, with the first value being the
3488 extension identifier and the second being an array of returned
3489 values. The array elements SHOULD be derived from (key,value)
3490 pairings for each location attribute that the authenticator
3491 supports. The following is an example of authenticator data
3492 where the returned array is comprised of a {longitude, latitude,
3493 altitude} triplet, following the coordinate representation
3494 defined in The W3C Geolocation API Specification.
3495
3496 ... -- RP ID hash (32 bytes)
3497 81 -- TUP and ED set
3498 00 00 00 01 -- (initial) signature counter
3499 ... -- all public key alg etc.
3500 A1 -- extension: CBOR map of one elemen
3501 t
3502 63 -- Value 1: CBOR text string of 3 by
3503 tes
3504 6C 6F 63 -- "loc" [=UTF-8 encoded=] string
3505 86 -- Value 2: array of 6 elements
3506 68 -- Element 1: CBOR text string of 8 bytes
3507 6C 61 74 69 74 75 64 65 -- "latitude" [=UTF-8 encoded=] stri
3508 ng
3509 FB ... -- Element 2: Latitude as CBOR encoded double-p
3510 recision float
3511 69 -- Element 3: CBOR text string of 9 bytes
3512 6C 6F 6E 67 69 74 75 64 65 -- "longitude" [=UTF-8 encoded=] str
3513 ing
3514 FB ... -- Element 4: Longitude as CBOR encoded double-
3515 precision float
3516 68 -- Element 5: CBOR text string of 8 bytes
3517 61 6C 74 69 74 75 64 65 -- "altitude" [=UTF-8 encoded=] stri
3518 ng
3519 FB ... -- Element 6: Altitude as CBOR encoded double-p
3520 recision float
3521
3522 9.8. User Verification Method Extension (uvm)
3523
3524 This registration extension and authentication extension enables use of
3525 a user verification method.
3526
3527 Extension identifier
3528 uvm
3529
3530 Client extension input
3531 The Boolean value true to indicate that this extension is

3636 requested by the Relying Party.
3637
3638 Client extension processing
3639 None, except creating the authenticator extension input from the
3640 client extension input.
3641
3642 Client extension output
3643 Returns a JSON object that encodes the location information in
3644 the authenticator extension output as a Coordinates value, as
3645 defined by The W3C Geolocation API Specification.
3646
3647 Authenticator extension input
3648 The Boolean value true, encoded in CBOR (major type 7, value
3649 21).
3650
3651 Authenticator extension processing
3652 If the authenticator does not support the extension, then the
3653 authenticator MUST ignore the extension request. If the
3654 authenticator accepts the extension, then the authenticator
3655 SHOULD only add this extension data to a packed attestation or
3656 assertion.
3657
3658 Authenticator extension output
3659 If the authenticator accepts the extension request, then
3660 authenticator extension output SHOULD provide location data in
3661 the form of a CBOR-encoded map, with the first value being the
3662 extension identifier and the second being an array of returned
3663 values. The array elements SHOULD be derived from (key,value)
3664 pairings for each location attribute that the authenticator
3665 supports. The following is an example of authenticator data
3666 where the returned array is comprised of a {longitude, latitude,
3667 altitude} triplet, following the coordinate representation
3668 defined in The W3C Geolocation API Specification.
3669
3670 ... -- [=RP ID=] hash (32 bytes)
3671 81 -- UP and ED set
3672 00 00 00 01 -- (initial) signature counter
3673 ... -- all public key alg etc.
3674 A1 -- extension: CBOR map of one elemen
3675 t
3676 63 -- Value 1: CBOR text string of 3 by
3677 tes
3678 6C 6F 63 -- "loc" [=UTF-8 encoded=] string
3679 86 -- Value 2: array of 6 elements
3680 68 -- Element 1: CBOR text string of 8 bytes
3681 6C 61 74 69 74 75 64 65 -- "latitude" [=UTF-8 encoded=] stri
3682 ng
3683 FB ... -- Element 2: Latitude as CBOR encoded double-p
3684 recision float
3685 69 -- Element 3: CBOR text string of 9 bytes
3686 6C 6F 6E 67 69 74 75 64 65 -- "longitude" [=UTF-8 encoded=] str
3687 ing
3688 FB ... -- Element 4: Longitude as CBOR encoded double-
3689 precision float
3690 68 -- Element 5: CBOR text string of 8 bytes
3691 61 6C 74 69 74 75 64 65 -- "altitude" [=UTF-8 encoded=] stri
3692 ng
3693 FB ... -- Element 6: Altitude as CBOR encoded double-p
3694 recision float
3695
3696 9.8. User Verification Method Extension (uvm)
3697
3698 This registration extension and authentication extension enables use of
3699 a user verification method.
3700
3701 Extension identifier
3702 uvm
3703
3704 Client extension input
The Boolean value true to indicate that this extension is

3532 requested by the WebAuthn Relying Party.
3533
3534 Client extension processing
3535 None, except creating the authenticator extension input from the
3536 client extension input.
3537
3538 Client extension output
3539 Returns a JSON array of 3-element arrays of numbers that encodes
3540 the factors in the authenticator extension output
3541
3542 Authenticator extension input
3543 The Boolean value true, encoded in CBOR (major type 7, value
3544 21).
3545
3546 Authenticator extension processing
3547 The authenticator sets the authenticator extension output to be
3548 a user verification index indicating the method used by the user
3549 to authorize the operation, as defined below. This extension can
3550 be added to attestation objects and assertions.
3551
3552 Authenticator extension output
3553 Authenticators can report up to 3 different user verification
3554 methods (factors) used in a single authentication instance,
3555 using the CBOR syntax defined below:
3556
3557 uvmFormat = [1*3 uvmEntry]
3558 uvmEntry = [
3559 userVerificationMethod: uint .size 4,
3560 keyProtectionType: uint .size 2,
3561 matcherProtectionType: uint .size 2
3562]
3563
3564 The semantics of the fields in each uvmEntry are as follows:
3565
3566 userVerificationMethod
3567 The authentication method/factor used by the authenticator
3568 to verify the user. Available values are defined in
3569 [FIDOReg], "User Verification Methods" section.
3570
3571 keyProtectionType
3572 The method used by the authenticator to protect the FIDO
3573 registration private key material. Available values are
3574 defined in [FIDOReg], "Key Protection Types" section.
3575
3576 matcherProtectionType
3577 The method used by the authenticator to protect the
3578 matcher that performs user verification. Available values
3579 are defined in [FIDOReg], "Matcher Protection Types"
3580 section.
3581
3582 If >3 factors can be used in an authentication instance the
3583 authenticator vendor must select the 3 factors it believes will
3584 be most relevant to the Server to include in the UVM.
3585
3586 Example for authenticator data containing one UVM extension for
3587 a multi-factor authentication instance where 2 factors were
3588 used:
3589
3590 ... -- RP ID hash (32 bytes)
3591 81 -- UP and ED set
3592 00 00 00 01 -- (initial) signature counter
3593 ... -- all public key alg etc.
3594 A1 -- extension: CBOR map of one element
3595 63 -- Key 1: CBOR text string of 3 bytes
3596 75 76 6d -- "uvm" [=UTF-8 encoded=] string
3597 82 -- Value 1: CBOR array of length 2 indicating two factor
3598 usage
3599 83 -- Item 1: CBOR array of length 3
3600 02 -- Subitem 1: CBOR integer for User Verification Method
3601 Fingerprint

3706 requested by the WebAuthn Relying Party.
3707
3708 Client extension processing
3709 None, except creating the authenticator extension input from the
3710 client extension input.
3711
3712 Client extension output
3713 Returns a JSON array of 3-element arrays of numbers that encodes
3714 the factors in the authenticator extension output
3715
3716 Authenticator extension input
3717 The Boolean value true, encoded in CBOR (major type 7, value
3718 21).
3719
3720 Authenticator extension processing
3721 The authenticator sets the authenticator extension output to be
3722 a user verification index indicating the method used by the user
3723 to authorize the operation, as defined below. This extension can
3724 be added to attestation objects and assertions.
3725
3726 Authenticator extension output
3727 Authenticators can report up to 3 different user verification
3728 methods (factors) used in a single authentication instance,
3729 using the CBOR syntax defined below:
3730
3731 uvmFormat = [1*3 uvmEntry]
3732 uvmEntry = [
3733 userVerificationMethod: uint .size 4,
3734 keyProtectionType: uint .size 2,
3735 matcherProtectionType: uint .size 2
3736]
3737
3738 The semantics of the fields in each uvmEntry are as follows:
3739
3740 userVerificationMethod
3741 The authentication method/factor used by the authenticator
3742 to verify the user. Available values are defined in
3743 [FIDOReg], "User Verification Methods" section.
3744
3745 keyProtectionType
3746 The method used by the authenticator to protect the FIDO
3747 registration private key material. Available values are
3748 defined in [FIDOReg], "Key Protection Types" section.
3749
3750 matcherProtectionType
3751 The method used by the authenticator to protect the
3752 matcher that performs user verification. Available values
3753 are defined in [FIDOReg], "Matcher Protection Types"
3754 section.
3755
3756 If >3 factors can be used in an authentication instance the
3757 authenticator vendor must select the 3 factors it believes will
3758 be most relevant to the Server to include in the UVM.
3759
3760 Example for authenticator data containing one UVM extension for
3761 a multi-factor authentication instance where 2 factors were
3762 used:
3763
3764 ... -- [=RP ID=] hash (32 bytes)
3765 81 -- UP and ED set
3766 00 00 00 01 -- (initial) signature counter
3767 ... -- all public key alg etc.
3768 A1 -- extension: CBOR map of one element
3769 63 -- Key 1: CBOR text string of 3 bytes
3770 75 76 6d -- "uvm" [=UTF-8 encoded=] string
3771 82 -- Value 1: CBOR array of length 2 indicating two factor
3772 usage
3773 83 -- Item 1: CBOR array of length 3
3774 02 -- Subitem 1: CBOR integer for User Verification Method
Fingerprint

3602	04	-- Subitem 2: CBOR short for Key Protection Type TEE
3603	02	-- Subitem 3: CBOR short for Matcher Protection Type TE
3604	E	
3605	83	-- Item 2: CBOR array of length 3
3606	04	-- Subitem 1: CBOR integer for User Verification Method
3607	Passcode	
3608	01	-- Subitem 2: CBOR short for Key Protection Type Softwa
3609	re	
3610	01	-- Subitem 3: CBOR short for Matcher Protection Type So
3611	ftware	
3612		
3613	10. IANA Considerations	
3614		
3615	10.1. WebAuthn Attestation Statement Format Identifier Registrations	
3616		
3617	This section registers the attestation statement formats defined in	
3618	Section 7 Defined Attestation Statement Formats in the IANA "WebAuthn	
3619	Attestation Statement Format Identifier" registry established by	
3620	[WebAuthn-Registries].	
3621	* WebAuthn Attestation Statement Format Identifier: packed	
3622	* Description: The "packed" attestation statement format is a	
3623	WebAuthn-optimized format for attestation data. It uses a very	
3624	compact but still extensible encoding method. This format is	
3625	implementable by authenticators with limited resources (e.g.,	
3626	secure elements).	
3627	* Specification Document: Section 7.2 Packed Attestation Statement	
3628	Format of this specification	
3629	* WebAuthn Attestation Statement Format Identifier: tpm	
3630	* Description: The TPM attestation statement format returns an	
3631	attestation statement in the same format as the packed attestation	
3632	statement format, although the the rawData and signature fields are	
3633	computed differently.	
3634	* Specification Document: Section 7.3 TPM Attestation Statement	
3635	Format of this specification	
3636	* WebAuthn Attestation Statement Format Identifier: android-key	
3637	* Description: Platform-provided authenticators based on Android	
3638	versions "N", and later, may provide this proprietary "hardware	
3639	attestation" statement.	
3640	* Specification Document: Section 7.4 Android Key Attestation	
3641	Statement Format of this specification	
3642	* WebAuthn Attestation Statement Format Identifier: android-safetynet	
3643	* Description: Android-based, platform-provided authenticators may	
3644	produce an attestation statement based on the Android SafetyNet	
3645	API.	
3646	* Specification Document: Section 7.5 Android SafetyNet Attestation	
3647	Statement Format of this specification	
3648	* WebAuthn Attestation Statement Format Identifier: fido-u2f	
3649	* Description: Used with FIDO U2F authenticators	
3650	* Specification Document: Section 7.6 FIDO U2F Attestation Statement	
3651	Format of this specification	
3652		
3653	10.2. WebAuthn Extension Identifier Registrations	
3654		
3655	This section registers the extension identifier values defined in	
3656	Section 8 WebAuthn Extensions in the IANA "WebAuthn Extension	
3657	Identifier" registry established by [WebAuthn-Registries].	
3658	* WebAuthn Extension Identifier: appid	
3659	* Description: This authentication extension allows Relying Parties	
3660	that have previously registered a credential using the legacy FIDO	
3661	JavaScript APIs to request an assertion.	
3662	* Specification Document: Section 9.1 FIDO AppId Extension (appid)	
3663	of this specification	
3664	* WebAuthn Extension Identifier: txAuthSimple	
3665	* Description: This registration extension and authentication	
3666	extension allows for a simple form of transaction authorization. A	
3667	WebAuthn Relying Party can specify a prompt string, intended for	
3668	display on a trusted device on the authenticator	
3669	* Specification Document: Section 9.2 Simple Transaction	
3670	Authorization Extension (txAuthSimple) of this specification	
3671	* WebAuthn Extension Identifier: txAuthGeneric	

3775	04	-- Subitem 2: CBOR short for Key Protection Type TEE
3776	02	-- Subitem 3: CBOR short for Matcher Protection Type TE
3777	E	
3778	83	-- Item 2: CBOR array of length 3
3779	04	-- Subitem 1: CBOR integer for User Verification Method
3780	Passcode	
3781	01	-- Subitem 2: CBOR short for Key Protection Type Softwa
3782	re	
3783	01	-- Subitem 3: CBOR short for Matcher Protection Type So
3784	ftware	
3785		
3786	10. IANA Considerations	
3787		
3788	10.1. WebAuthn Attestation Statement Format Identifier Registrations	
3789		
3790	This section registers the attestation statement formats defined in	
3791	Section 7 Defined Attestation Statement Formats in the IANA "WebAuthn	
3792	Attestation Statement Format Identifier" registry established by	
3793	[WebAuthn-Registries].	
3794	* WebAuthn Attestation Statement Format Identifier: packed	
3795	* Description: The "packed" attestation statement format is a	
3796	WebAuthn-optimized format for attestation data. It uses a very	
3797	compact but still extensible encoding method. This format is	
3798	implementable by authenticators with limited resources (e.g.,	
3799	secure elements).	
3800	* Specification Document: Section 7.2 Packed Attestation Statement	
3801	Format of this specification	
3802	* WebAuthn Attestation Statement Format Identifier: tpm	
3803	* Description: The TPM attestation statement format returns an	
3804	attestation statement in the same format as the packed attestation	
3805	statement format, although the the rawData and signature fields are	
3806	computed differently.	
3807	* Specification Document: Section 7.3 TPM Attestation Statement	
3808	Format of this specification	
3809	* WebAuthn Attestation Statement Format Identifier: android-key	
3810	* Description: Platform-provided authenticators based on Android	
3811	versions "N", and later, may provide this proprietary "hardware	
3812	attestation" statement.	
3813	* Specification Document: Section 7.4 Android Key Attestation	
3814	Statement Format of this specification	
3815	* WebAuthn Attestation Statement Format Identifier: android-safetynet	
3816	* Description: Android-based, platform-provided authenticators may	
3817	produce an attestation statement based on the Android SafetyNet	
3818	API.	
3819	* Specification Document: Section 7.5 Android SafetyNet Attestation	
3820	Statement Format of this specification	
3821	* WebAuthn Attestation Statement Format Identifier: fido-u2f	
3822	* Description: Used with FIDO U2F authenticators	
3823	* Specification Document: Section 7.6 FIDO U2F Attestation Statement	
3824	Format of this specification	
3825		
3826	10.2. WebAuthn Extension Identifier Registrations	
3827		
3828	This section registers the extension identifier values defined in	
3829	Section 8 WebAuthn Extensions in the IANA "WebAuthn Extension	
3830	Identifier" registry established by [WebAuthn-Registries].	
3831	* WebAuthn Extension Identifier: appid	
3832	* Description: This authentication extension allows Relying Parties	
3833	that have previously registered a credential using the legacy FIDO	
3834	JavaScript APIs to request an assertion.	
3835	* Specification Document: Section 9.1 FIDO AppId Extension (appid)	
3836	of this specification	
3837	* WebAuthn Extension Identifier: txAuthSimple	
3838	* Description: This registration extension and authentication	
3839	extension allows for a simple form of transaction authorization. A	
3840	WebAuthn Relying Party can specify a prompt string, intended for	
3841	display on a trusted device on the authenticator	
3842	* Specification Document: Section 9.2 Simple Transaction	
3843	Authorization Extension (txAuthSimple) of this specification	
3844	* WebAuthn Extension Identifier: txAuthGeneric	

3672 * Description: This registration extension and authentication
3673 extension allows images to be used as transaction authorization
3674 prompts as well. This allows authenticators without a font
3675 rendering engine to be used and also supports a richer visual
3676 appearance than accomplished with the webauthn.txauth.simple
3677 extension.
3678 * Specification Document: Section 9.3 Generic Transaction
3679 Authorization Extension (txAuthGeneric) of this specification
3680 * WebAuthn Extension Identifier: authnSel
3681 * Description: This registration extension allows a WebAuthn Relying
3682 Party to guide the selection of the authenticator that will be
3683 leveraged when creating the credential. It is intended primarily
3684 for WebAuthn Relying Parties that wish to tightly control the
3685 experience around credential creation.
3686 * Specification Document: Section 9.4 Authenticator Selection
3687 Extension (authnSel) of this specification
3688 * WebAuthn Extension Identifier: exts
3689 * Description: This registration extension enables the Relying Party
3690 to determine which extensions the authenticator supports. The
3691 extension data is a list (CBOR array) of extension identifiers
3692 encoded as UTF-8 Strings. This extension is added automatically by
3693 the authenticator. This extension can be added to attestation
3694 statements.
3695 * Specification Document: Section 9.5 Supported Extensions Extension
3696 (exts) of this specification
3697 * WebAuthn Extension Identifier: uvi
3698 * Description: This registration extension and authentication
3699 extension enables use of a user verification index. The user
3700 verification index is a value uniquely identifying a user
3701 verification data record. The UVI data can be used by servers to
3702 understand whether an authentication was authorized by the exact
3703 same biometric data as the initial key generation. This allows the
3704 detection and prevention of "friendly fraud".
3705 * Specification Document: Section 9.6 User Verification Index
3706 Extension (uvi) of this specification
3707 * WebAuthn Extension Identifier: loc
3708 * Description: The location registration extension and authentication
3709 extension provides the client device's current location to the
3710 WebAuthn relying party, if supported by the client device and
3711 subject to user consent.
3712 * Specification Document: Section 9.7 Location Extension (loc) of
3713 this specification
3714 * WebAuthn Extension Identifier: uvm
3715 * Description: This registration extension and authentication
3716 extension enables use of a user verification method. The user
3717 verification method extension returns to the Webauthn relying party
3718 which user verification methods (factors) were used for the
3719 WebAuthn operation.
3720 * Specification Document: Section 9.8 User Verification Method
3721 Extension (uvm) of this specification
3722

3845 * Description: This registration extension and authentication
3846 extension allows images to be used as transaction authorization
3847 prompts as well. This allows authenticators without a font
3848 rendering engine to be used and also supports a richer visual
3849 appearance than accomplished with the webauthn.txauth.simple
3850 extension.
3851 * Specification Document: Section 9.3 Generic Transaction
3852 Authorization Extension (txAuthGeneric) of this specification
3853 * WebAuthn Extension Identifier: authnSel
3854 * Description: This registration extension allows a WebAuthn Relying
3855 Party to guide the selection of the authenticator that will be
3856 leveraged when creating the credential. It is intended primarily
3857 for WebAuthn Relying Parties that wish to tightly control the
3858 experience around credential creation.
3859 * Specification Document: Section 9.4 Authenticator Selection
3860 Extension (authnSel) of this specification
3861 * WebAuthn Extension Identifier: exts
3862 * Description: This registration extension enables the Relying Party
3863 to determine which extensions the authenticator supports. The
3864 extension data is a list (CBOR array) of extension identifiers
3865 encoded as UTF-8 Strings. This extension is added automatically by
3866 the authenticator. This extension can be added to attestation
3867 statements.
3868 * Specification Document: Section 9.5 Supported Extensions Extension
3869 (exts) of this specification
3870 * WebAuthn Extension Identifier: uvi
3871 * Description: This registration extension and authentication
3872 extension enables use of a user verification index. The user
3873 verification index is a value uniquely identifying a user
3874 verification data record. The UVI data can be used by servers to
3875 understand whether an authentication was authorized by the exact
3876 same biometric data as the initial key generation. This allows the
3877 detection and prevention of "friendly fraud".
3878 * Specification Document: Section 9.6 User Verification Index
3879 Extension (uvi) of this specification
3880 * WebAuthn Extension Identifier: loc
3881 * Description: The location registration extension and authentication
3882 extension provides the client device's current location to the
3883 WebAuthn relying party, if supported by the client device and
3884 subject to user consent.
3885 * Specification Document: Section 9.7 Location Extension (loc) of
3886 this specification
3887 * WebAuthn Extension Identifier: uvm
3888 * Description: This registration extension and authentication
3889 extension enables use of a user verification method. The user
3890 verification method extension returns to the Webauthn relying party
3891 which user verification methods (factors) were used for the
3892 WebAuthn operation.
3893 * Specification Document: Section 9.8 User Verification Method
3894 Extension (uvm) of this specification
3895

10.3. COSE Algorithm Registrations

This section registers identifiers for RSASSA-PKCS1-v1_5 [RFC8017] algorithms using SHA-2 hash functions in the IANA COSE Algorithms registry [IANA-COSE-ALGS-REG].

* Name: RS256
* Value: -257
* Description: RSASSA-PKCS1-v1_5 w/ SHA-256
* Reference: Section 8.2 of [RFC8017]
* Recommended: No
* Name: RS384
* Value: -258
* Description: RSASSA-PKCS1-v1_5 w/ SHA-384
* Reference: Section 8.2 of [RFC8017]
* Recommended: No
* Name: RS512
* Value: -259
* Description: RSASSA-PKCS1-v1_5 w/ SHA-512
* Reference: Section 8.2 of [RFC8017]

3723 11. Sample scenarios
3724
3725 This section is not normative.
3726
3727 In this section, we walk through some events in the lifecycle of a
3728 public key credential, along with the corresponding sample code for
3729 using this API. Note that this is an example flow, and does not limit
3730 the scope of how the API can be used.
3731
3732 As was the case in earlier sections, this flow focuses on a use case
3733 involving an external first-factor authenticator with its own display.
3734 One example of such an authenticator would be a smart phone. Other
3735 authenticator types are also supported by this API, subject to
3736 implementation by the platform. For instance, this flow also works
3737 without modification for the case of an authenticator that is embedded
3738 in the client platform. The flow also works for the case of an
3739 authenticator without its own display (similar to a smart card) subject
3740 to specific implementation considerations. Specifically, the client
3741 platform needs to display any prompts that would otherwise be shown by
3742 the authenticator, and the authenticator needs to allow the client
3743 platform to enumerate all the authenticator's credentials so that the
3744 client can have information to show appropriate prompts.
3745
3746 11.1. Registration
3747
3748 This is the first-time flow, in which a new credential is created and
3749 registered with the server.
3750
3751 1. The user visits example.com, which serves up a script. At this
3752 point, the user **must** already be logged in using a legacy username
3753 and password, or additional authenticator, or other means
acceptable to the Relying Party.
3754
3755 2. The Relying Party script runs the code snippet below.
3756 3. The client platform searches for and locates the authenticator.
3757 4. The client platform connects to the authenticator, performing any
pairing actions if necessary.
3758 5. The authenticator shows appropriate UI for the user to select the
3759 authenticator on which the new credential will be created, and
3760 obtains a biometric or other authorization gesture from the user.
3761 6. The authenticator returns a response to the client platform, which
3762 in turn returns a response to the Relying Party script. If the user
3763 declined to select an authenticator or provide authorization, an
3764 appropriate error is returned.
3765 7. If a new credential was created,
3766 + The Relying Party script sends the newly generated credential
3767 public key to the server, along with additional information
3768 such as attestation regarding the provenance and
3769 characteristics of the authenticator.
3770 + The server stores the credential public key in its database
3771 and associates it with the user as well as with the
3772 characteristics of authentication indicated by attestation,
3773 also storing a friendly name for later use.
3774 + The script may store data such as the credential ID in local
3775 storage, to improve future UX by narrowing the choice of
3776 credential for the user.
3777
3778 The sample code for generating and registering a new key follows:
3779 if (!PublicKeyCredential) { /* Platform not capable. Handle error. */ }
3780
3781 var publicKey = {
3782 challenge: Uint8Array.from(window.atob("PGifxAoBwCkWkm4b1Cill5otCphilh6MijdbW
3783 FjomA="), c=>c.charCodeAt(0)),
3784
3785 // Relying Party:
3786 rp: {
3787 name: "Acme"
3788 },

3915 * Recommended: No
3916
3917 11. Sample scenarios
3918
3919 This section is not normative.
3920
3921 In this section, we walk through some events in the lifecycle of a
3922 public key credential, along with the corresponding sample code for
3923 using this API. Note that this is an example flow, and does not limit
3924 the scope of how the API can be used.
3925
3926 As was the case in earlier sections, this flow focuses on a use case
3927 involving an external first-factor authenticator with its own display.
3928 One example of such an authenticator would be a smart phone. Other
3929 authenticator types are also supported by this API, subject to
3930 implementation by the platform. For instance, this flow also works
3931 without modification for the case of an authenticator that is embedded
3932 in the client platform. The flow also works for the case of an
3933 authenticator without its own display (similar to a smart card) subject
3934 to specific implementation considerations. Specifically, the client
3935 platform needs to display any prompts that would otherwise be shown by
3936 the authenticator, and the authenticator needs to allow the client
3937 platform to enumerate all the authenticator's credentials so that the
3938 client can have information to show appropriate prompts.
3939
3940 11.1. Registration
3941
3942 This is the first-time flow, in which a new credential is created and
3943 registered with the server. **In this flow, the Relying Party does not**
3944 **have a preference for platform authenticator or roaming authenticators.**
3945 1. The user visits example.com, which serves up a script. At this
3946 point, the user **may** already be logged in using a legacy username
3947 and password, or additional authenticator, or other means
3948 acceptable to the Relying Party. **Or the user may be in the process**
3949 **of creating a new account.**
3950 2. The Relying Party script runs the code snippet below.
3951 3. The client platform searches for and locates the authenticator.
3952 4. The client platform connects to the authenticator, performing any
3953 pairing actions if necessary.
3954 5. The authenticator shows appropriate UI for the user to select the
3955 authenticator on which the new credential will be created, and
3956 obtains a biometric or other authorization gesture from the user.
3957 6. The authenticator returns a response to the client platform, which
3958 in turn returns a response to the Relying Party script. If the user
3959 declined to select an authenticator or provide authorization, an
3960 appropriate error is returned.
3961 7. If a new credential was created,
3962 + The Relying Party script sends the newly generated credential
3963 public key to the server, along with additional information
3964 such as attestation regarding the provenance and
3965 characteristics of the authenticator.
3966 + The server stores the credential public key in its database
3967 and associates it with the user as well as with the
3968 characteristics of authentication indicated by attestation,
3969 also storing a friendly name for later use.
3970 + The script may store data such as the credential ID in local
3971 storage, to improve future UX by narrowing the choice of
3972 credential for the user.
3973
3974 The sample code for generating and registering a new key follows:
3975 if (!PublicKeyCredential) { /* Platform not capable. Handle error. */ }
3976
3977 var publicKey = {
3978 challenge: Uint8Array.from(window.atob("PGifxAoBwCkWkm4b1Cill5otCphilh6MijdbW
3979 FjomA="), c=>c.charCodeAt(0)),
3980
3981 // Relying Party:
3982 rp: {
3983 name: "Acme"
3984 },

```

3988 // User:
3989 user: {
3990   id: "1098237235409872"
3991   name: "john.p.smith@example.com",
3992   displayName: "John P. Smith",
3993   icon: "https://pics.acme.com/00/p/aBjjpqPb.png"
3994 },
3995 // This Relying Party will accept either an ES256 or RS256 credential, but
3996 // prefers an ES256 credential.
3997 pubKeyCredParams: [
3998   {
3999     type: "public-key",
4000     alg: -7 // "ES256" as registered in the IANA COSE Algorithms registry
4001   },
4002   {
4003     type: "public-key",
4004     alg: -257 // Value registered by this specification for "RS256"
4005   }
4006 ],
4007 timeout: 60000, // 1 minute
4008 excludeCredentials: [], // No exclude list of PKCredDescriptors
4009 extensions: {"webauthn.location": true} // Include location information
4010 // in attestation
4011 };
4012
4013 // Note: The following call will cause the authenticator to display UI.
4014 navigator.credentials.create({ publicKey })
4015   .then(function (newCredentialInfo) {
4016     // Send new credential info to server for verification and registration.
4017   }).catch(function (err) {
4018     // No acceptable authenticator or user refused consent. Handle appropriately
4019   });
4020
4021
4022

```

```

4024 This is flow for when the Relying Party is specifically interested in
4025 creating a public key credential with a platform authenticator.
4026 1. The user visits example.com and clicks on the login button, which
4027 redirects the user to login.example.com.
4028 2. The user enters a username and password to log in. After successful
4029 login, the user is redirected back to example.com.
4030 3. The Relying Party script runs the code snippet below.
4031 4. The user agent asks the user whether they are willing to register
4032 with the Relying Party using an available platform authenticator.
4033 5. If the user is not willing, terminate this flow.
4034 6. The user is shown appropriate UI and guided in creating a
4035 credential using one of the available platform authenticators. Upon
4036 successful credential creation, the RP script conveys the new
4037 credential to the server.
4038 if (!PublicKeyCredential) { /* Platform not capable of the API. Handle error. */
4039 }
4040
4041 PublicKeyCredential.isPlatformAuthenticatorAvailable()
4042 .then(function (userIntent) {
4043
4044     // If the user has affirmed willingness to register with RP using an available platform authenticator
4045     if (userIntent) {
4046         var publicKeyOptions = { /* Public key credential creation options.
4047
4048 */};
4049
4050         // Create and register credentials.
4051         return navigator.credentials.create({ "publicKey": publicKeyOptions
4052 });
4053     } else {
4054

```



```

3827
3828 This is the flow when a user with an already registered credential
3829 visits a website and wants to authenticate using the credential.
3830 1. The user visits example.com, which serves up a script.
3831 2. The script asks the client platform for an Authentication
3832 Assertion, providing as much information as possible to narrow the
3833 choice of acceptable credentials for the user. This may be obtained
3834 from the data that was stored locally after registration, or by
3835 other means such as prompting the user for a username.
3836 3. The Relying Party script runs one of the code snippets below.
3837 4. The client platform searches for and locates the authenticator.
3838 5. The client platform connects to the authenticator, performing any
3839 pairing actions if necessary.
3840 6. The authenticator presents the user with a notification that their
3841 attention is required. On opening the notification, the user is
3842 shown a friendly selection menu of acceptable credentials using the
3843 account information provided when creating the credentials, along
3844 with some information on the origin that is requesting these keys.
3845 7. The authenticator obtains a biometric or other authorization
3846 gesture from the user.
3847 8. The authenticator returns a response to the client platform, which
3848 in turn returns a response to the Relying Party script. If the user
3849 declined to select a credential or provide an authorization, an
3850 appropriate error is returned.
3851 9. If an assertion was successfully generated and returned,
3852 + The script sends the assertion to the server.
3853 + The server examines the assertion, extracts the credential ID,
3854 looks up the registered credential public key it is database,
3855 and verifies the assertion's authentication signature. If
3856 valid, it looks up the identity associated with the
3857 assertion's credential ID; that identity is now authenticated.
3858 If the credential ID is not recognized by the server (e.g., it
3859 has been deregistered due to inactivity) then the
3860 authentication has failed; each Relying Party will handle this
3861 in its own way.
3862 + The server now does whatever it would otherwise do upon
3863 successful authentication -- return a success page, set
3864 authentication cookies, etc.
3865
3866 If the Relying Party script does not have any hints available (e.g.,
3867 from locally stored data) to help it narrow the list of credentials,
3868 then the sample code for performing such an authentication might look
3869 like this:
3870 if (!PublicKeyCredential) { /* Platform not capable. Handle error. */ }
3871
3872 var options = {
3873   challenge: new TextEncoder().encode("climb a mountain"),
3874   timeout: 60000, // 1 minute
3875   allowList: [{ type: "public-key" }]
3876 };
3877
3878 navigator.credentials.get({ "publicKey": options })
3879   .then(function (assertion) {
3880     // Send assertion to server for verification
3881   }).catch(function (err) {
3882     // No acceptable credential or user refused consent. Handle appropriately.
3883   });
3884

```

```

4055 // Record that the user does not intend to use a platform authentica
4056 tor
4057 // and default the user to a password-based flow in the future.
4058 }
4059
4060 }).then(function (newCredentialInfo) {
4061   // Send new credential info to server for verification and registration.
4062 }).catch( function(err) {
4063   // Something went wrong. Handle appropriately.
4064 });
4065

```

11.3. Authentication

```

4066 This is the flow when a user with an already registered credential
4067 visits a website and wants to authenticate using the credential.
4068 1. The user visits example.com, which serves up a script.
4069 2. The script asks the client platform for an Authentication
4070 Assertion, providing as much information as possible to narrow the
4071 choice of acceptable credentials for the user. This may be obtained
4072 from the data that was stored locally after registration, or by
4073 other means such as prompting the user for a username.
4074 3. The Relying Party script runs one of the code snippets below.
4075 4. The client platform searches for and locates the authenticator.
4076 5. The client platform connects to the authenticator, performing any
4077 pairing actions if necessary.
4078 6. The authenticator presents the user with a notification that their
4079 attention is required. On opening the notification, the user is
4080 shown a friendly selection menu of acceptable credentials using the
4081 account information provided when creating the credentials, along
4082 with some information on the origin that is requesting these keys.
4083 7. The authenticator obtains a biometric or other authorization
4084 gesture from the user.
4085 8. The authenticator returns a response to the client platform, which
4086 in turn returns a response to the Relying Party script. If the user
4087 declined to select a credential or provide an authorization, an
4088 appropriate error is returned.
4089 9. If an assertion was successfully generated and returned,
4090 + The script sends the assertion to the server.
4091 + The server examines the assertion, extracts the credential ID,
4092 looks up the registered credential public key it is database,
4093 and verifies the assertion's authentication signature. If
4094 valid, it looks up the identity associated with the
4095 assertion's credential ID; that identity is now authenticated.
4096 If the credential ID is not recognized by the server (e.g., it
4097 has been deregistered due to inactivity) then the
4098 authentication has failed; each Relying Party will handle this
4099 in its own way.
4100 + The server now does whatever it would otherwise do upon
4101 successful authentication -- return a success page, set
4102 authentication cookies, etc.
4103
4104 If the Relying Party script does not have any hints available (e.g.,
4105 from locally stored data) to help it narrow the list of credentials,
4106 then the sample code for performing such an authentication might look
4107 like this:
4108 if (!PublicKeyCredential) { /* Platform not capable. Handle error. */ }
4109
4110 var options = {
4111   challenge: new TextEncoder().encode("climb a mountain"),
4112   timeout: 60000, // 1 minute
4113   allowCredentials: [{ type: "public-key" }]
4114 };
4115
4116 navigator.credentials.get({ "publicKey": options })
4117   .then(function (assertion) {
4118     // Send assertion to server for verification
4119   }).catch(function (err) {
4120     // No acceptable credential or user refused consent. Handle appropriately.
4121   });
4122

```

```
3885 On the other hand, if the Relying Party script has some hints to help
3886 it narrow the list of credentials, then the sample code for performing
3887 such an authentication might look like the following. Note that this
3888 sample also demonstrates how to use the extension for transaction
3889 authorization.
3890 if (!PublicKeyCredential) { /* Platform not capable. Handle error. */ }
3891
3892 var encoder = new TextEncoder();
3893 var acceptableCredential1 = {
3894   type: "public-key",
3895   id: encoder.encode("!!!!!!hi there!!!!!!\n")
3896 };
3897 var acceptableCredential2 = {
3898   type: "public-key",
3899   id: encoder.encode("roses are red, violets are blue\n")
3900 };
3901
3902 var options = {
3903   challenge: encoder.encode("climb a mountain"),
3904   timeout: 60000, // 1 minute
3905   allowList: [acceptableCredential1, acceptableCredential2];
3906
3907   extensions: { 'webauthn.txauth.simple':
3908     "Wave your hands in the air like you just don't care" };
3909 };
3910
3911 navigator.credentials.get({ "publicKey": options })
3912   .then(function (assertion) {
3913     // Send assertion to server for verification
3914   }).catch(function (err) {
3915     // No acceptable credential or user refused consent. Handle appropriately.
3916   });
```

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: Richard Barnes, Dominic Battr, Domenic Denicola, Rahul Ghosh, Brad Hill, Jing Jin, Angelo Liao, Anne van Kesteren, Ian Kilpatrick, Giridhar Mandyam, Axel Nennker, Kimberly

```
4125 On the other hand, if the Relying Party script has some hints to help
4126 it narrow the list of credentials, then the sample code for performing
4127 such an authentication might look like the following. Note that this
4128 sample also demonstrates how to use the extension for transaction
4129 authorization.
4130 if (!PublicKeyCredential) { /* Platform not capable. Handle error. */ }
4131
4132 var encoder = new TextEncoder();
4133 var acceptableCredential1 = {
4134   type: "public-key",
4135   id: encoder.encode("!!!!!!hi there!!!!!!\n")
4136 };
4137 var acceptableCredential2 = {
4138   type: "public-key",
4139   id: encoder.encode("roses are red, violets are blue\n")
4140 };
4141
4142 var options = {
4143   challenge: encoder.encode("climb a mountain"),
4144   timeout: 60000, // 1 minute
4145   allowCredentials: [acceptableCredential1, acceptableCredential2]
4146 };
4147
4148   extensions: { 'webauthn.txauth.simple':
4149     "Wave your hands in the air like you just don't care" };
4150 };
4151
4152 navigator.credentials.get({ "publicKey": options })
4153   .then(function (assertion) {
4154     // Send assertion to server for verification
4155   }).catch(function (err) {
4156     // No acceptable credential or user refused consent. Handle appropriately.
4157   });
```

11.4. 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: Richard Barnes, Dominic Battr, Domenic Denicola, Rahul Ghosh, Brad Hill, Jing Jin, Angelo Liao, Anne van Kesteren, Ian Kilpatrick, Giridhar Mandyam, Axel Nennker, Kimberly

3954	Paulhamus, Adam Powers, Yaron Sheffer, Mike West, Jeffrey Yasskin,
3955	Boris Zbarsky.
3956	
3957	Index
3958	
3959	Terms defined by this specification
3960	
3961	* AAGUID, in 9.4
3962	* algorithm, in 4.3
3963	* allowList, in 4.6
3964	* Assertion, in 3
3965	* assertion signature, in 5
3966	* Attachment, in 4.5.3
3967	* attachment, in 4.5.2
3968	* attachment modality, in 4.5.3
3969	* Attestation, in 3
3970	* Attestation Certificate, in 3
3971	* Attestation data, in 5.3.1
3972	* Attestation information, in 3
3973	* attestation key pair, in 3
3974	* attestationObject, in 4.2.1
3975	* attestation objects, in 3
3976	* attestation private key, in 3
3977	* attestation public key, in 3
3978	* attestation signature, in 5
3979	
3980	* attestation statement format, in 5.3
3981	* attestation statement format identifier, in 7.1
3982	* attestation type, in 5.3
3983	* Authentication, in 3
3984	* Authentication Assertion, in 3
3985	* authentication extension, in 8
3986	* AuthenticationExtensions
3987	+ definition of, in 4.7
3988	+ (typedef), in 4.7
3989	* Authenticator, in 3
3990	* AuthenticatorAssertionResponse, in 4.2.2
3991	
3992	* AuthenticatorAttestationResponse, in 4.2.1
3993	* authenticatorCancel, in 5.2.3
3994	* authenticator data, in 5.1
3995	* authenticatorData, in 4.2.2
3996	* authenticator data claimed to have been used for the attestation,
3997	in 5.3.2
3998	* authenticator data for the attestation, in 5.3.2
3999	* authenticator extension, in 8
4000	* authenticator extension input, in 8.3
4001	* authenticator extension output, in 8.5
4002	* Authenticator extension processing, in 8.5
4003	* authenticatorExtensions, in 4.8.1
4004	* authenticatorGetAssertion, in 5.2.2
4005	* authenticatorMakeCredential, in 5.2.1
4006	* AuthenticatorResponse, in 4.2
4007	* authenticatorSelection, in 4.5
4008	* AuthenticatorSelectionCriteria, in 4.5.2
4009	* AuthenticatorSelectionList, in 9.4
4010	
4011	* Authorization Gesture, in 3
4012	* Base64url Encoding, in 2.1
4013	* Basic Attestation, in 5.3.3
4014	* Biometric Recognition, in 3
4015	* ble, in 4.8.4
4016	* CBOR, in 2.1
4017	* Ceremony, in 3
4018	* challenge
4019	+ dict-member for MakeCredentialOptions, in 4.5
4020	+ dict-member for PublicKeyCredentialRequestOptions, in 4.6
4021	+ dict-member for CollectedClientData, in 4.8.1
4022	* Client, in 3

4195	Paulhamus, Adam Powers, Yaron Sheffer, Mike West, Jeffrey Yasskin,
4196	Boris Zbarsky.
4197	
4198	Index
4199	
4200	Terms defined by this specification
4201	
4202	* aa, in 4.4.3
4203	* AAGUID, in 9.4
4204	* alg, in 4.3
4205	* allowCredentials, in 4.5
4206	* Assertion, in 3
4207	* assertion signature, in 5
4208	* attachment modality, in 4.4.4
4209	
4210	* Attestation, in 3
4211	* Attestation Certificate, in 3
4212	* Attestation data, in 5.3.1
4213	
4214	* attestation key pair, in 3
4215	* attestationObject, in 4.2.1
4216	* attestation object, in 5.3
4217	* attestation private key, in 3
4218	* attestation public key, in 3
4219	* attestation signature, in 5
4220	* attestation statement, in 5.3
4221	* attestation statement format, in 5.3
4222	* attestation statement format identifier, in 7.1
4223	* attestation type, in 5.3
4224	* Authentication, in 3
4225	* Authentication Assertion, in 3
4226	* authentication extension, in 8
4227	* AuthenticationExtensions
4228	+ definition of, in 4.6
4229	+ (typedef), in 4.6
4230	* Authenticator, in 3
4231	* AuthenticatorAssertionResponse, in 4.2.2
4232	* AuthenticatorAttachment, in 4.4.4
4233	* AuthenticatorAttestationResponse, in 4.2.1
4234	* authenticatorCancel, in 5.2.3
4235	* authenticator data, in 5.1
4236	* authenticatorData, in 4.2.2
4237	* authenticator data claimed to have been used for the attestation,
4238	in 5.3.2
4239	* authenticator data for the attestation, in 5.3.2
4240	* authenticator extension, in 8
4241	* authenticator extension input, in 8.3
4242	* authenticator extension output, in 8.5
4243	* Authenticator extension processing, in 8.5
4244	* authenticatorExtensions, in 4.7.1
4245	* authenticatorGetAssertion, in 5.2.2
4246	* authenticatorMakeCredential, in 5.2.1
4247	* AuthenticatorResponse, in 4.2
4248	* authenticatorSelection, in 4.4
4249	* AuthenticatorSelectionCriteria, in 4.4.3
4250	* AuthenticatorSelectionList, in 9.4
4251	* AuthenticatorTransport, in 4.7.4
4252	* Authorization Gesture, in 3
4253	* Base64url Encoding, in 2.1
4254	* Basic Attestation, in 5.3.3
4255	* Biometric Recognition, in 3
4256	* ble, in 4.7.4
4257	* CBOR, in 2.1
4258	* Ceremony, in 3
4259	* challenge
4260	+ dict-member for MakePublicKeyCredentialOptions, in 4.4
4261	+ dict-member for PublicKeyCredentialRequestOptions, in 4.5
	+ dict-member for CollectedClientData, in 4.7.1
	* Client, in 3

4020 * client data, in 4.8.1
4021 * clientDataJSON, in 4.2
4022 * client extension, in 8
4023 * client extension input, in 8.3
4024 * client extension output, in 8.4
4025 * Client extension processing, in 8.4
4026 * clientExtensionResults, in 4.1
4027 * clientExtensions, in 4.8.1
4028 * Client-Side, in 3
4029 * client-side credential private key storage, in 3
4030 * Client-side-resident Credential Private Key, in 3
4031 * CollectedClientData, in 4.8.1
4032 * Conforming User Agent, in 3

4033 * [[Create]](options), in 4.1.3
4034 * credential key pair, in 3
4035 * credential private key, in 3
4036 * Credential Public Key, in 3
4037 * [CredentialRequestOptions](#), in 4.1.1
4038 * cross-platform, in 4.5.3
4039 * "cross-platform", in 4.5.3
4040 * cross-platform attached, in 4.5.3
4041 * cross-platform attachment, in 4.5.3
4042 * DAA, in 5.3.3
4043 * [[DiscoverFromExternalSource]](options), in 4.1.4
4044 * [[discovery]], in 4.1
4045 * displayName, in 4.4
4046 * ECDAA, in 5.3.3
4047 * ECDAA-Issuer public key, in 7.2
4048 * Elliptic Curve based Direct Anonymous Attestation, in 5.3.3
4049 * excludeList, in 4.5
4050 * extension identifier, in 8.1
4051 * extensions
4052 * + dict-member for MakeCredentialOptions, in 4.5
4053 * + dict-member for PublicKeyCredentialRequestOptions, in 4.6
4054 * [ExternalTransport](#), in 4.8.4
4055 * hashAlg, in 4.8.1
4056 * Hash of the serialized client data, in 4.8.1
4057 * icon, in 4.5.1
4058 * id
4059 * + dict-member for PublicKeyCredentialEntity, in 4.5.1
4060 * + dict-member for PublicKeyCredentialDescriptor, in 4.8.3
4061 * [[identifier]], in 4.1
4062 * identifier of the ECDAA-Issuer public key, in 7.2
4063 * [JSON-serialized client data](#), in 4.8.1
4064 * [MakeCredentialOptions](#), in 4.5
4065 * name, in 4.5.1
4066 * nfc, in 4.8.4
4067 * origin, in 4.8.1
4068 * parameters, in 4.5
4069 * platform, in 4.5.3
4070 * "platform", in 4.5.3
4071 * platform attachment, in 4.5.3
4072 * platform authenticators, in 4.5.3
4073 * Privacy CA, in 5.3.3

4074 * publicKey
4075 * + dict-member for CredentialRequestOptions, in 4.1.1
4076 * + dict-member for CredentialCreationOptions, in 4.1.2
4077 * public-key, in 4.8.2
4078 * Public Key Credential, in 3
4079 * PublicKeyCredential, in 4.1
4080 * PublicKeyCredentialDescriptor, in 4.8.3
4081 * PublicKeyCredentialEntity, in 4.5.1
4082 * PublicKeyCredentialParameters, in 4.3
4083 * PublicKeyCredentialRequestOptions, in 4.6
4084 * PublicKeyCredentialType, in 4.8.2
4085 * PublicKeyCredentialUserEntity, in 4.4

4262 * client data, in 4.7.1
4263 * clientDataJSON, in 4.2
4264 * client extension, in 8
4265 * client extension input, in 8.3
4266 * client extension output, in 8.4
4267 * Client extension processing, in 8.4
4268 * clientExtensionResults, in 4.1
4269 * clientExtensions, in 4.7.1
4270 * Client-Side, in 3
4271 * client-side credential private key storage, in 3
4272 * Client-side-resident Credential Private Key, in 3
4273 * CollectedClientData, in 4.7.1
4274 * Conforming User Agent, in 3
4275 * [COSEAlgorithmIdentifier](#)
4276 * + definition of, in 4.7.5
4277 * + (typedef), in 4.7.5
4278 * [[Create]](options), in 4.1.3
4279 * credential key pair, in 3
4280 * credential private key, in 3
4281 * Credential Public Key, in 3
4282 * cross-platform attached, in 4.4.4
4283 * cross-platform attachment, in 4.4.4

4284 * DAA, in 5.3.3
4285 * [[DiscoverFromExternalSource]](options), in 4.1.4
4286 * [[discovery]], in 4.1
4287 * displayName, in 4.4.2
4288 * ECDAA, in 5.3.3
4289 * ECDAA-Issuer public key, in 7.2
4290 * Elliptic Curve based Direct Anonymous Attestation, in 5.3.3
4291 * excludeCredentials, in 4.4
4292 * extension identifier, in 8.1
4293 * extensions
4294 * + dict-member for MakePublicKeyCredentialOptions, in 4.4
4295 * + dict-member for PublicKeyCredentialRequestOptions, in 4.5
4296 * hashAlgorithm, in 4.7.1
4297 * Hash of the serialized client data, in 4.7.1
4298 * icon, in 4.4.1

4299 * id
4300 * + dict-member for PublicKeyCredentialEntity, in 4.4.1
4301 * + dict-member for PublicKeyCredentialDescriptor, in 4.7.3
4302 * [[identifier]], in 4.1
4303 * identifier of the ECDAA-Issuer public key, in 7.2
4304 * isPlatformAuthenticatorAvailable(), in 4.1.5
4305 * [JSON-serialized client data](#), in 4.7.1
4306 * [MakePublicKeyCredentialOptions](#), in 4.4
4307 * name, in 4.4.1
4308 * nfc, in 4.7.4
4309 * origin, in 4.7.1
4310 * "plat", in 4.4.4
4311 * plat, in 4.4.4
4312 * platform attachment, in 4.4.4
4313 * platform authenticators, in 4.4.4
4314 * Privacy CA, in 5.3.3
4315 * [pubKeyCredParams](#), in 4.4
4316 * publicKey
4317 * + dict-member for CredentialCreationOptions, in 4.1.1
4318 * + dict-member for CredentialRequestOptions, in 4.1.2
4319 * public-key, in 4.7.2
4320 * Public Key Credential, in 3
4321 * PublicKeyCredential, in 4.1
4322 * PublicKeyCredentialDescriptor, in 4.7.3
4323 * PublicKeyCredentialEntity, in 4.4.1
4324 * PublicKeyCredentialParameters, in 4.3
4325 * PublicKeyCredentialRequestOptions, in 4.5
4326 * PublicKeyCredentialType, in 4.7.2
4327 * PublicKeyCredentialUserEntity, in 4.4.2

4086 * rawId, in 4.1
4087 * Registration, in 3
4088 * registration extension, in 8
4089 * Relying Party, in 3
4090 * Relying Party Identifier, in 3
4091 * requireResidentKey, in 4.5.2
4092 * response, in 4.1
4093 * roaming authenticators, in 4.5.3
4094 * rp, in 4.5
4095 * rpId, in 4.6

4096 * RP ID, in 3
4097 * Self Attestation, in 5.3.3
4098 * signature, in 4.2.2

4099 * Test of User Presence, in 3
4100 * timeout
4101 + dict-member for MakeCredentialOptions, in 4.5
4102 + dict-member for PublicKeyCredentialRequestOptions, in 4.6
4103 * tokenBinding, in 4.8.1
4104 * Transport, in 4.8.4
4105 * transports, in 4.8.3
4106 * TUP, in 3
4107 * [[type]], in 4.1
4108 * type
4109 + dict-member for PublicKeyCredentialParameters, in 4.3
4110 + dict-member for PublicKeyCredentialDescriptor, in 4.8.3
4111 * usb, in 4.8.4
4112 * user, in 4.5

4113 * User Consent, in 3

4114 * User Verification, in 3
4115 * User Verified, in 3

4116 * Web Authentication API, in 4
4117 * WebAuthn Client, in 3

4118 Terms defined by reference
4119
4120
4121 * [CREDENTIAL-MANAGEMENT-1] defines the following terms:
4122 + CredentialCreationOptions

4123 + create()

4124 * [ECMAScript] defines the following terms:
4125 + %arraybuffer%
4126 + internal slot
4127 + stringify
4128 * [ENCODING] defines the following terms:
4129 + utf-8 encode
4130 * [HTML] defines the following terms:
4131 + ascii serialization of an origin
4132 + dom manipulation task source
4133 + effective domain
4134 + global object

4328 * Rate Limiting, in 3
4329 * rawId, in 4.1
4330 * Registration, in 3
4331 * registration extension, in 8
4332 * Relying Party, in 3
4333 * Relying Party Identifier, in 3

4334 * response, in 4.1
4335 * rk, in 4.4.3
4336 * roaming authenticators, in 4.4.4
4337 * rp, in 4.4
4338 * rpId, in 4.5
4339 * RP ID, in 3
4340 * Self Attestation, in 5.3.3
4341 * signature, in 4.2.2
4342 * Signing procedure, in 5.3.2
4343 * Test of User Presence, in 3
4344 * timeout
4345 + dict-member for MakePublicKeyCredentialOptions, in 4.4
4346 + dict-member for PublicKeyCredentialRequestOptions, in 4.5
4347 * tokenBindingId, in 4.7.1
4348 * transports, in 4.7.3

4349 * [[type]], in 4.1
4350 * type
4351 + dict-member for PublicKeyCredentialParameters, in 4.3
4352 + dict-member for PublicKeyCredentialDescriptor, in 4.7.3
4353 * UP, in 3
4354 * usb, in 4.7.4
4355 * user, in 4.4
4356 * User Consent, in 3
4357 * User Present, in 3
4358 * User Verification, in 3
4359 * User Verified, in 3
4360 * UV, in 3
4361 * uv, in 4.4.3
4362 * Verification procedures, in 5.3.2
4363 * Web Authentication API, in 4
4364 * WebAuthn Client, in 3
4365 * "xplat", in 4.4.4
4366 * xplat, in 4.4.4

4367 Terms defined by reference
4368
4369
4370 * [CREDENTIAL-MANAGEMENT-1] defines the following terms:
4371 + Credential
4372 + CredentialCreationOptions
4373 + CredentialRequestOptions
4374 + CredentialsContainer
4375 + [[CollectFromCredentialStore]](options)
4376 + [[Store]](credential)
4377 + [[discovery]]
4378 + [[type]]
4379 + create()
4380 + get()
4381 + id
4382 + remote
4383 + type
4384 * [ECMAScript] defines the following terms:
4385 + %arraybuffer%
4386 + internal slot
4387 + stringify
4388 * [ENCODING] defines the following terms:
4389 + utf-8 encode
4390 * [HTML] defines the following terms:
4391 + ascii serialization of an origin
4392 + dom manipulation task source
4393 + effective domain
4394 + global object

4135 + in parallel
4136 + is a registrable domain suffix of or is equal to
4137 + is not a registrable domain suffix of and is not equal to
4138 + origin

4139 + relevant settings object
4140 + task
4141 + task source
4142 + **unicode serialization of an origin**
4143 * [HTML52] defines the following terms:

4144 + opaque origin
4145 + origin
4146 * [INFRA] defines the following terms:
4147 + append (for list)
4148 + append (for set)
4149 + continue
4150 + for each (for list)
4151 + for each (for map)
4152 + is empty
4153 + is not empty
4154 + item
4155 + list
4156 + map
4157 + ordered set
4158 + remove
4159 + set
4160 * [secure-contexts] defines the following terms:
4161 + secure context
4162 * [TokenBinding] defines the following terms:
4163 + token binding
4164 + token binding id
4165 * [URL] defines the following terms:

4166 + url serializer
4167 * **[webappsec-credential-management-1] defines the following terms:**
4168 + **Credential**
4169 + **CredentialsContainer**
4170 + **[[CollectFromCredentialStore]](options)**
4171 + **[[Store]](credential)**
4172 + **[[discovery]]**
4173 + **[[type]]**
4174 + **get()**
4175 + **id**
4176 + **remote**
4177 + **type**
4178 * [WebCryptoAPI] defines the following terms:
4179 + **AlgorithmIdentifier**
4180 + **normalizing an algorithm**
4181 + recognized algorithm name
4182 * [WebIDL] defines the following terms:
4183 + ArrayBuffer
4184 + BufferSource
4185 + ConstraintError
4186 + DOMException
4187 + DOMString
4188 + NotAllowedError
4189 + NotFoundError
4190 + NotSupportedError
4191 + Promise

4192 + SecureContext
4193 + SecurityError
4194 + TypeError
4195 + USVString

4395 + in parallel
4396 + is a registrable domain suffix of or is equal to
4397 + is not a registrable domain suffix of and is not equal to
4398 + origin
4399 + **promise**
4400 + relevant settings object
4401 + task
4402 + task source

4403 * [HTML52] defines the following terms:
4404 + **document.domain**
4405 + opaque origin
4406 + origin
4407 * [INFRA] defines the following terms:
4408 + append (for list)
4409 + append (for set)
4410 + continue
4411 + for each (for list)
4412 + for each (for map)
4413 + is empty
4414 + is not empty
4415 + item
4416 + list
4417 + map
4418 + ordered set
4419 + remove
4420 + set
4421 * [secure-contexts] defines the following terms:
4422 + secure context
4423 * [TokenBinding] defines the following terms:
4424 + token binding
4425 + token binding id
4426 * [URL] defines the following terms:
4427 + **domain**
4428 + **empty host**
4429 + **host**
4430 + **ipv4 address**
4431 + **ipv6 address**
4432 + **opaque host**
4433 + url serializer
4434 + **valid domain**
4435 + **valid domain string**

4436 * [WebCryptoAPI] defines the following terms:

4437 + recognized algorithm name
4438 * [WebIDL] defines the following terms:
4439 + ArrayBuffer
4440 + BufferSource
4441 + ConstraintError
4442 + DOMException
4443 + DOMString
4444 + NotAllowedError
4445 + NotFoundError
4446 + NotSupportedError
4447 + Promise
4448 + **SameObject**
4449 + SecureContext
4450 + SecurityError
4451 + TypeError
4452 + USVString

4196 + boolean
4197 + dictionary
4198 + interface object

4199 + present
4200 + simple exception
4201 + unsigned long
4202
4203 References
4204
4205 Normative References
4206
4207 [CDDL]
4208 C. Vigano; H. Birkholz. CBOR data definition language (CDDL): a
4209 notational convention to express CBOR data structures. 21
4210 September 2016. Internet Draft (work in progress). URL:
4211 <https://tools.ietf.org/html/draft-greevenbosch-appsawg-cbor-cddl>
4212
4213 [CREDENTIAL-MANAGEMENT-1]
4214 Mike West. Credential Management Level 1. URL:
4215 <https://www.w3.org/TR/credential-management-1/>
4216
4217 [DOM4]
4218 Anne van Kesteren. DOM Standard. Living Standard. URL:
4219 <https://dom.spec.whatwg.org/>
4220
4221 [ECMAScript]
4222 ECMAScript Language Specification. URL:
4223 <https://tc39.github.io/ecma262/>
4224
4225 [ENCODING]
4226 Anne van Kesteren. Encoding Standard. Living Standard. URL:
4227 <https://encoding.spec.whatwg.org/>
4228
4229 [FIDOEcdaaAlgorithm]
4230 R. Lindemann; et al. FIDO ECDA Algorithm. FIDO Alliance
4231 Implementation Draft. URL:
4232 <https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-ecdaa-algorithm-v1.1-id-20170202.html>
4233
4234 [FIDOReg]
4235 R. Lindemann; D. Baghdasaryan; B. Hill. FIDO UAF Registry of
4236 Predefined Values. FIDO Alliance Proposed Standard. URL:
4237 <https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-reg-v1.0-ps-20141208.html>
4238
4239 [HTML]
4240 Anne van Kesteren; et al. HTML Standard. Living Standard. URL:
4241 <https://html.spec.whatwg.org/multipage/>
4242
4243 [HTML52]
4244 Steve Faulkner; et al. HTML 5.2. URL:
4245 <https://www.w3.org/TR/html52/>
4246
4247
4248

4249 [INFRA]
4250 Anne van Kesteren; Domenic Denicola. Infra Standard. Living
4251 Standard. URL: <https://infra.spec.whatwg.org/>
4252
4253 [RFC2119]
4254 S. Bradner. Key words for use in RFCs to Indicate Requirement
4255 Levels. March 1997. Best Current Practice. URL:
4256 <https://tools.ietf.org/html/rfc2119>
4257

4453 + UnknownError
4454 + Unscopable
4455 + boolean

4456 + interface object
4457 + long
4458 + present
4459 + simple exception
4460 + unsigned long
4461
4462 References
4463
4464 Normative References
4465
4466 [CDDL]
4467 C. Vigano; H. Birkholz. CBOR data definition language (CDDL): a
4468 notational convention to express CBOR data structures. 21
4469 September 2016. Internet Draft (work in progress). URL:
4470 <https://tools.ietf.org/html/draft-greevenbosch-appsawg-cbor-cddl>
4471
4472 [CREDENTIAL-MANAGEMENT-1]
4473 Mike West. Credential Management Level 1. URL:
4474 <https://www.w3.org/TR/credential-management-1/>
4475
4476 [DOM4]
4477 Anne van Kesteren. DOM Standard. Living Standard. URL:
4478 <https://dom.spec.whatwg.org/>
4479
4480 [ECMAScript]
4481 ECMAScript Language Specification. URL:
4482 <https://tc39.github.io/ecma262/>
4483
4484 [ENCODING]
4485 Anne van Kesteren. Encoding Standard. Living Standard. URL:
4486 <https://encoding.spec.whatwg.org/>
4487
4488 [FIDOEcdaaAlgorithm]
4489 R. Lindemann; et al. FIDO ECDA Algorithm. FIDO Alliance
4490 Implementation Draft. URL:
4491 <https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-ecdaa-algorithm-v1.1-id-20170202.html>
4492
4493 [FIDOReg]
4494 R. Lindemann; D. Baghdasaryan; B. Hill. FIDO UAF Registry of
4495 Predefined Values. FIDO Alliance Proposed Standard. URL:
4496 <https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-reg-v1.0-ps-20141208.html>
4497
4498 [HTML]
4499 Anne van Kesteren; et al. HTML Standard. Living Standard. URL:
4500 <https://html.spec.whatwg.org/multipage/>
4501
4502 [HTML52]
4503 Steve Faulkner; et al. HTML 5.2. URL:
4504 <https://www.w3.org/TR/html52/>
4505
4506
4507
4508 [IANA-COSE-ALGS-REG]
4509 IANA CBOR Object Signing and Encryption (COSE) Algorithms
4510 Registry. URL:
4511 <https://www.iana.org/assignments/cose/cose.xhtml#algorithms>
4512
4513 [INFRA]
4514 Anne van Kesteren; Domenic Denicola. Infra Standard. Living
4515 Standard. URL: <https://infra.spec.whatwg.org/>
4516
4517 [RFC2119]
4518 S. Bradner. Key words for use in RFCs to Indicate Requirement
4519 Levels. March 1997. Best Current Practice. URL:
4520 <https://tools.ietf.org/html/rfc2119>
4521

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

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

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

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

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

[SECURE-CONTEXTS]
Mike West. Secure Contexts. URL:
<https://www.w3.org/TR/secure-contexts/>

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

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

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

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

[WebCryptoAPI]
Mark Watson. Web Cryptography API. URL:
<https://www.w3.org/TR/WebCryptoAPI/>

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

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

Informative References

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

[FIDO-APPID]
D. Balfanz; et al. FIDO AppID and Facets. FIDO Alliance Review
Draft. URL:

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

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

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

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

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

[SECURE-CONTEXTS]
Mike West. Secure Contexts. URL:
<https://www.w3.org/TR/secure-contexts/>

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

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

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

[WebCryptoAPI]
Mark Watson. Web Cryptography API. URL:
<https://www.w3.org/TR/WebCryptoAPI/>

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

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

Informative References

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

[FIDO-APPID]
D. Balfanz; et al. FIDO AppID and Facets. FIDO Alliance Review
Draft. URL:

4328 https://fidoalliance.org/specs/fido-uaf-v1.1-rd-20161005/fido-ap
4329 pid-and-facets-v1.1-rd-20161005.html
4330
4331 [FIDO-U2F-Message-Formats]
4332 D. Balfanz; J. Ehrensvar; J. Lang. FIDO U2F Raw Message
4333 Formats. FIDO Alliance Implementation Draft. URL:
4334 https://fidoalliance.org/specs/fido-u2f-v1.1-id-20160915/fido-u2
4335 f-raw-message-formats-v1.1-id-20160915.html
4336
4337 [FIDOMetadataService]
4338 R. Lindemann; B. Hill; D. Baghdasaryan. FIDO Metadata Service
4339 v1.0. FIDO Alliance Proposed Standard. URL:
4340 https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-ua
4341 f-metadata-service-v1.0-ps-20141208.html
4342
4343 [FIDOSecRef]
4344 R. Lindemann; D. Baghdasaryan; B. Hill. FIDO Security Reference.
4345 FIDO Alliance Proposed Standard. URL:
4346 https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-se
4347 curity-ref-v1.0-ps-20141208.html
4348
4349 [GeoJSON]
4350 The GeoJSON Format Specification. URL:
4351 http://geojson.org/geojson-spec.html
4352
4353 [ISOBiometricVocabulary]
4354 ISO/IEC JTC1/SC37. Information technology -- Vocabulary --
4355 Biometrics. 15 December 2012. International Standard: ISO/IEC
4356 2382-37:2012(E) First Edition. URL:
4357 http://standards.iso.org/ittf/PubliclyAvailableStandards/c055194
4358 _ISOIEC_2382-37_2012.zip
4359
4360 [RFC4949]
4361 R. Shirey. Internet Security Glossary, Version 2. August 2007.
4362 Informational. URL: https://tools.ietf.org/html/rfc4949
4363
4364 [RFC5280]
4365 D. Cooper; et al. Internet X.509 Public Key Infrastructure
4366 Certificate and Certificate Revocation List (CRL) Profile. May
4367 2008. Proposed Standard. URL:
4368 https://tools.ietf.org/html/rfc5280
4369
4370 [RFC6454]
4371 A. Barth. The Web Origin Concept. December 2011. Proposed
4372 Standard. URL: https://tools.ietf.org/html/rfc6454
4373
4374 [RFC7515]
4375 M. Jones; J. Bradley; N. Sakimura. JSON Web Signature (JWS). May
4376 2015. Proposed Standard. URL:
4377 https://tools.ietf.org/html/rfc7515
4378
4379 [TPMv2-EK-Profile]
4380 TCG EK Credential Profile for TPM Family 2.0. URL:
4381 http://www.trustedcomputinggroup.org/wp-content/uploads/Credenti
4382 al_Profile_EK_V2.0_R14_published.pdf
4383
4384 [TPMv2-Part1]
4385 Trusted Platform Module Library, Part 1: Architecture. URL:
4386 http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-
4387 2.0-Part-1-Architecture-01.38.pdf
4388

4588 https://fidoalliance.org/specs/fido-uaf-v1.1-rd-20161005/fido-ap
4589 pid-and-facets-v1.1-rd-20161005.html
4590
4591 [FIDO-U2F-Message-Formats]
4592 D. Balfanz; J. Ehrensvar; J. Lang. FIDO U2F Raw Message
4593 Formats. FIDO Alliance Implementation Draft. URL:
4594 https://fidoalliance.org/specs/fido-u2f-v1.1-id-20160915/fido-u2
4595 f-raw-message-formats-v1.1-id-20160915.html
4596
4597 [FIDOMetadataService]
4598 R. Lindemann; B. Hill; D. Baghdasaryan. FIDO Metadata Service
4599 v1.0. FIDO Alliance Proposed Standard. URL:
4600 https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-ua
4601 f-metadata-service-v1.0-ps-20141208.html
4602
4603 [FIDOSecRef]
4604 R. Lindemann; D. Baghdasaryan; B. Hill. FIDO Security Reference.
4605 FIDO Alliance Proposed Standard. URL:
4606 https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-se
4607 curity-ref-v1.0-ps-20141208.html
4608
4609 [GeoJSON]
4610 The GeoJSON Format Specification. URL:
4611 http://geojson.org/geojson-spec.html
4612
4613 [ISOBiometricVocabulary]
4614 ISO/IEC JTC1/SC37. Information technology -- Vocabulary --
4615 Biometrics. 15 December 2012. International Standard: ISO/IEC
4616 2382-37:2012(E) First Edition. URL:
4617 http://standards.iso.org/ittf/PubliclyAvailableStandards/c055194
4618 _ISOIEC_2382-37_2012.zip
4619
4620 [RFC4949]
4621 R. Shirey. Internet Security Glossary, Version 2. August 2007.
4622 Informational. URL: https://tools.ietf.org/html/rfc4949
4623
4624 [RFC5280]
4625 D. Cooper; et al. Internet X.509 Public Key Infrastructure
4626 Certificate and Certificate Revocation List (CRL) Profile. May
4627 2008. Proposed Standard. URL:
4628 https://tools.ietf.org/html/rfc5280
4629
4630 [RFC6265]
4631 A. Barth. HTTP State Management Mechanism. April 2011. Proposed
4632 Standard. URL: https://tools.ietf.org/html/rfc6265
4633
4634 [RFC6454]
4635 A. Barth. The Web Origin Concept. December 2011. Proposed
4636 Standard. URL: https://tools.ietf.org/html/rfc6454
4637
4638 [RFC7515]
4639 M. Jones; J. Bradley; N. Sakimura. JSON Web Signature (JWS). May
4640 2015. Proposed Standard. URL:
4641 https://tools.ietf.org/html/rfc7515
4642
4643 [RFC8017]
4644 K. Moriarty, Ed.; et al. PKCS #1: RSA Cryptography
4645 Specifications Version 2.2. November 2016. Informational. URL:
4646 https://tools.ietf.org/html/rfc8017
4647
4648 [TPMv2-EK-Profile]
4649 TCG EK Credential Profile for TPM Family 2.0. URL:
4650 http://www.trustedcomputinggroup.org/wp-content/uploads/Credenti
4651 al_Profile_EK_V2.0_R14_published.pdf
4652
4653 [TPMv2-Part1]
4654 Trusted Platform Module Library, Part 1: Architecture. URL:
4655 http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-
4656 2.0-Part-1-Architecture-01.38.pdf
4657


```
4389 [TPMv2-Part2]
4390 Trusted Platform Module Library, Part 2: Structures. URL:
4391 http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-
4392 2.0-Part-2-Structures-01.38.pdf
4393
4394 [TPMv2-Part3]
4395 Trusted Platform Module Library, Part 3: Commands. URL:
4396 http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-
4397 2.0-Part-3-Commands-01.38.pdf
4398
4399 [UAFProtocol]
4400 R. Lindemann; et al. FIDO UAF Protocol Specification v1.0. FIDO
4401 Alliance Proposed Standard. URL:
4402 https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-ua
4403 f-protocol-v1.0-ps-20141208.html
4404
4405 IDL Index
4406
4407 [SecureContext]
4408 interface PublicKeyCredential : Credential {
4409     readonly attribute ArrayBuffer rawId;
4410     readonly attribute AuthenticatorResponse response;
4411     readonly attribute AuthenticationExtensions clientExtensionResults;
4412 };
4413
4414 partial dictionary CredentialRequestOptions {
4415     PublicKeyCredentialRequestOptions? publicKey;
4416 };
4417
4418 partial dictionary CredentialCreationOptions {
4419     MakeCredentialOptions? publicKey;
4420 };
4421
4422 [SecureContext]
4423 interface AuthenticatorResponse {
4424     readonly attribute ArrayBuffer clientDataJSON;
4425 };
4426
4427 [SecureContext]
4428 interface AuthenticatorAttestationResponse : AuthenticatorResponse {
4429     readonly attribute ArrayBuffer attestationObject;
4430 };
4431
4432 [SecureContext]
4433 interface AuthenticatorAssertionResponse : AuthenticatorResponse {
4434     readonly attribute ArrayBuffer authenticatorData;
4435     readonly attribute ArrayBuffer signature;
4436 };
4437
4438 dictionary PublicKeyCredentialParameters {
4439     required PublicKeyCredentialType type;
4440     required AlgorithmIdentifier algorithm;
4441 };
4442
4443 dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
4444     DOMString displayName;
4445 };
4446
4447 dictionary MakeCredentialOptions {
4448     required PublicKeyCredentialEntity rp;
4449     required PublicKeyCredentialUserEntity user;
4450
4451     required BufferSource challenge;
4452     required sequence<PublicKeyCredentialParameters> parameters;
```

```
4659 [TPMv2-Part2]
4660 Trusted Platform Module Library, Part 2: Structures. URL:
4661 http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-
4662 2.0-Part-2-Structures-01.38.pdf
4663
4664 [TPMv2-Part3]
4665 Trusted Platform Module Library, Part 3: Commands. URL:
4666 http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-
4667 2.0-Part-3-Commands-01.38.pdf
4668
4669 [UAFProtocol]
4670 R. Lindemann; et al. FIDO UAF Protocol Specification v1.0. FIDO
4671 Alliance Proposed Standard. URL:
4672 https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-ua
4673 f-protocol-v1.0-ps-20141208.html
4674
4675 IDL Index
4676
4677 [SecureContext]
4678 interface PublicKeyCredential : Credential {
4679     [SameObject] readonly attribute ArrayBuffer rawId;
4680     [SameObject] readonly attribute AuthenticatorResponse response;
4681     [SameObject] readonly attribute AuthenticationExtensions clientExtensionResults;
4682 };
4683
4684 partial dictionary CredentialCreationOptions {
4685     MakePublicKeyCredentialOptions publicKey;
4686 };
4687
4688 partial dictionary CredentialRequestOptions {
4689     PublicKeyCredentialRequestOptions? publicKey;
4690 };
4691
4692 [SecureContext]
4693 partial interface PublicKeyCredential {
4694     [Unscopable] Promise < boolean > isPlatformAuthenticatorAvailable();
4695 };
4696
4697 [SecureContext]
4698 interface AuthenticatorResponse {
4699     [SameObject] readonly attribute ArrayBuffer clientDataJSON;
4700 };
4701
4702 [SecureContext]
4703 interface AuthenticatorAttestationResponse : AuthenticatorResponse {
4704     [SameObject] readonly attribute ArrayBuffer attestationObject;
4705 };
4706
4707 [SecureContext]
4708 interface AuthenticatorAssertionResponse : AuthenticatorResponse {
4709     [SameObject] readonly attribute ArrayBuffer authenticatorData;
4710     [SameObject] readonly attribute ArrayBuffer signature;
4711 };
4712
4713 dictionary PublicKeyCredentialParameters {
4714     required PublicKeyCredentialType type;
4715     required COSEAlgorithmIdentifier alg;
4716 };
4717
4718 dictionary MakePublicKeyCredentialOptions {
4719     required PublicKeyCredentialEntity rp;
4720     required PublicKeyCredentialUserEntity user;
4721
4722     required BufferSource challenge;
4723     required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
```

```
4453 unsigned long timeout;
4454 sequence<PublicKeyCredentialDescriptor> excludeList;
4455 AuthenticatorSelectionCriteria authenticatorSelection;
4456 AuthenticationExtensions extensions;
4457 };
4458
4459 dictionary PublicKeyCredentialEntity {
4460   DOMString id;
4461   DOMString name;
4462   USVString icon;
4463 };
4464
4465
4466 dictionary AuthenticatorSelectionCriteria {
4467   Attachment attachment;
4468   boolean requireResidentKey = false;
4469 };
4470
4471 enum Attachment {
4472   "platform",
4473   "cross-platform"
4474 };
4475
4476 dictionary PublicKeyCredentialRequestOptions {
4477   required BufferSource challenge;
4478   unsigned long timeout;
4479   USVString rpId;
4480   sequence<PublicKeyCredentialDescriptor> allowList = [];
4481   AuthenticationExtensions extensions;
4482 };
4483
4484 typedef record<DOMString, any> AuthenticationExtensions;
4485
4486 dictionary CollectedClientData {
4487   required DOMString challenge;
4488   required DOMString origin;
4489   required DOMString hashAlg;
4490   DOMString tokenBinding;
4491   AuthenticationExtensions clientExtensions;
4492   AuthenticationExtensions authenticatorExtensions;
4493 };
4494
4495 enum PublicKeyCredentialType {
4496   "public-key"
4497 };
4498
4499 dictionary PublicKeyCredentialDescriptor {
4500   required PublicKeyCredentialType type;
4501   required BufferSource id;
4502   sequence<Transport> transports;
4503 };
4504
4505 enum Transport {
4506   "usb",
4507   "nfc",
4508   "ble"
4509 };
4510
4511
4512
4513 typedef sequence<AAGUID> AuthenticatorSelectionList;
4514
4515 typedef BufferSource AAGUID;
```

```
4724 unsigned long timeout;
4725 sequence<PublicKeyCredentialDescriptor> excludeCredentials = [];
4726 AuthenticatorSelectionCriteria authenticatorSelection;
4727 AuthenticationExtensions extensions;
4728 };
4729
4730 dictionary PublicKeyCredentialEntity {
4731   DOMString id;
4732   DOMString name;
4733   USVString icon;
4734 };
4735
4736 dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
4737   DOMString displayName;
4738 };
4739
4740 dictionary AuthenticatorSelectionCriteria {
4741   AuthenticatorAttachment aa; // authenticatorAttachment
4742   boolean rk = false; // requireResidentKey
4743   boolean uv = false; // requireUserVerification
4744 };
4745
4746 enum AuthenticatorAttachment {
4747   "plat", // Platform attachment
4748   "xplat" // Cross-platform attachment
4749 };
4750
4751 dictionary PublicKeyCredentialRequestOptions {
4752   required BufferSource challenge;
4753   unsigned long timeout;
4754   USVString rpId;
4755   sequence<PublicKeyCredentialDescriptor> allowCredentials = [];
4756   AuthenticationExtensions extensions;
4757 };
4758
4759 typedef record<DOMString, any> AuthenticationExtensions;
4760
4761 dictionary CollectedClientData {
4762   required DOMString challenge;
4763   required DOMString origin;
4764   required DOMString hashAlgorithm;
4765   DOMString tokenBindingId;
4766   AuthenticationExtensions clientExtensions;
4767   AuthenticationExtensions authenticatorExtensions;
4768 };
4769
4770 enum PublicKeyCredentialType {
4771   "public-key"
4772 };
4773
4774 dictionary PublicKeyCredentialDescriptor {
4775   required PublicKeyCredentialType type;
4776   required BufferSource id;
4777   sequence<AuthenticatorTransport> transports;
4778 };
4779
4780 enum AuthenticatorTransport {
4781   "usb",
4782   "nfc",
4783   "ble"
4784 };
4785
4786
4787 typedef long COSEAlgorithmIdentifier;
4788
4789 typedef sequence<AAGUID> AuthenticatorSelectionList;
4790
4791 typedef BufferSource AAGUID;
4792
4793
```

```

4794 #base64url-encodingReferenced in:
4795 * 4.1. PublicKeyCredential Interface
4796 * 4.1.3. Create a new credential - PublicKeyCredential's
4797   [[Create]](options) method (2)
4798 * 4.1.4. Use an existing credential to make an assertion -
4799   PublicKeyCredential's [[DiscoverFromExternalSource]](options)
4800   method (2)
4801 * 6.2. Verifying an authentication assertion
4802
4803 #cborReferenced in:
4804 * 4.1.3. Create a new credential - PublicKeyCredential's
4805   [[Create]](options) method
4806 * 4.1.4. Use an existing credential to make an assertion -
4807   PublicKeyCredential's [[DiscoverFromExternalSource]](options)
4808   method
4809 * 5.1. Authenticator data (2)
4810 * 8. WebAuthn Extensions (2) (3)
4811 * 8.2. Defining extensions (2)
4812 * 8.3. Extending request parameters
4813 * 8.4. Client extension processing (2)
4814 * 8.5. Authenticator extension processing (2) (3) (4) (5)
4815
4816 #attestationReferenced in:
4817 * 3. Terminology
4818 * 5. WebAuthn Authenticator model (2)
4819 * 5.3. Attestation (2) (3) (4)
4820
4821 #attestation-certificateReferenced in:
4822 * 3. Terminology (2)
4823 * 7.3.1. TPM attestation statement certificate requirements
4824
4825 #attestation-key-pairReferenced in:
4826 * 3. Terminology (2)
4827 * 5.3. Attestation

```

4820 #attestation-certificateReferenced in:
4821 * 3. Terminology (2)
4822 * 7.3.1. TPM attestation statement certificate requirements
4823
4824 #attestation-key-pairReferenced in:
4825 * 3. Terminology (2)
4826 * 5.3. Attestation
4827
4828 #attestation-private-keyReferenced in:
4829 * 5. WebAuthn Authenticator model
4830 * 5.3. Attestation
4831
4832 #attestation-public-keyReferenced in:
4833 * 5.3. Attestation
4834
4835 #authenticationReferenced in:
4836 * 1. Introduction (2)
4837 * 3. Terminology (2) (3) (4) (5) (6) (7)
4838 * 6.2. Verifying an authentication assertion
4839
4840 #authentication-assertionReferenced in:
4841 * 1. Introduction
4842 * 3. Terminology (2) (3)
4843 * 4.1. PublicKeyCredential Interface
4844 * 4.2.2. Web Authentication Assertion (interface
4845 AuthenticatorAssertionResponse)
4846 * 4.5. Options for Assertion Generation (dictionary
4847 PublicKeyCredentialRequestOptions)
4848 * 8. WebAuthn Extensions
4849
4850 #authenticatorReferenced in:
4851 * 1. Introduction (2) (3) (4)
4852 * 1.1. Use Cases
4853 * 2. Conformance
4854 * 3. Terminology (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13)
4855 (14) (15)

4576 * 4.1. PublicKeyCredential Interface
4577 * 4.1.3. Create a new credential - PublicKeyCredential's
4578 [[Create]](options) method
4579 * 4.1.4. Use an existing credential -
4580 PublicKeyCredential::[[DiscoverFromExternalSource]](options) method

4581 * 4.2. Authenticator Responses (interface AuthenticatorResponse)
4582 * 4.2.1. Information about Public Key Credential (interface
4583 AuthenticatorAttestationResponse) (2)
4584 * 4.2.2. Web Authentication Assertion (interface
4585 AuthenticatorAssertionResponse)
4586 * 4.5.3. Credential Attachment enumeration (enum Attachment)
4587 * 4.6. Options for Assertion Generation (dictionary

4588 PublicKeyCredentialRequestOptions)
4589 * 5. WebAuthn Authenticator model
4590 * 5.1. Authenticator data
4591 * 5.3. Credential Attestation

4592 * 5.3.5.1. Privacy
4593 * 5.3.5.2. Attestation Certificate and Attestation Certificate CA
4594 Compromise
4595 * 6.1. Registering a new credential
4596 * 7.2. Packed Attestation Statement Format
4597 * 7.4. Android Key Attestation Statement Format
4598 * 7.5. Android SafetyNet Attestation Statement Format
4599 * 9.5. Supported Extensions Extension (exts)
4600 * 9.6. User Verification Index Extension (uvi)
4601 * 9.7. Location Extension (loc) (2) (3) (4)
4602 * 9.8. User Verification Method Extension (uvm)
4603 * 11. Sample scenarios
4604
4605 #authorization-gestureReferenced in:
4606 * 1.1.1. Registration
4607 * 1.1.2. Authentication
4608 * 1.1.3. Other use cases and configurations
4609 * 3. Terminology (2) (3) (4) (5) (6)
4610
4611 #biometric-recognitionReferenced in:
4612 * 3. Terminology (2)
4613
4614 #ceremonyReferenced in:
4615 * 1. Introduction
4616 * 3. Terminology (2) (3) (4) (5) (6) (7)
4617 * 6.1. Registering a new credential
4618 * 6.2. Verifying an authentication assertion
4619
4620 #clientReferenced in:
4621 * 3. Terminology

4622
4623 #conforming-user-agentReferenced in:
4624 * 1. Introduction
4625 * 2. Conformance (2) (3)
4626 * 3. Terminology (2)
4627
4628 #credential-public-keyReferenced in:

4857 * 4. Web Authentication API (2) (3)
4858 * 4.1. PublicKeyCredential Interface
4859 * 4.1.3. Create a new credential - PublicKeyCredential's
4860 [[Create]](options) method (2)
4861 * 4.1.4. Use an existing credential to make an assertion -
4862 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
4863 method (2) (3)
4864 * 4.2. Authenticator Responses (interface AuthenticatorResponse)
4865 * 4.2.1. Information about Public Key Credential (interface
4866 AuthenticatorAttestationResponse) (2)
4867 * 4.2.2. Web Authentication Assertion (interface
4868 AuthenticatorAssertionResponse)
4869 * 4.4.4. Authenticator Attachment enumeration (enum
4870 AuthenticatorAttachment)
4871 * 4.5. Options for Assertion Generation (dictionary
4872 PublicKeyCredentialRequestOptions)
4873 * 5. WebAuthn Authenticator model (2) (3) (4) (5) (6)
4874 * 5.1. Authenticator data
4875 * 5.2.1. The authenticatorMakeCredential operation
4876 * 5.2.2. The authenticatorGetAssertion operation (2) (3)
4877 * 5.3. Attestation (2) (3) (4) (5) (6) (7) (8) (9)
4878 * 5.3.2. Attestation Statement Formats
4879 * 5.3.4. Generating an Attestation Object (2)
4880 * 5.3.5.1. Privacy
4881 * 5.3.5.2. Attestation Certificate and Attestation Certificate CA
4882 Compromise
4883 * 6.1. Registering a new credential
4884 * 7.2. Packed Attestation Statement Format
4885 * 7.4. Android Key Attestation Statement Format
4886 * 7.5. Android SafetyNet Attestation Statement Format
4887 * 9.5. Supported Extensions Extension (exts)
4888 * 9.6. User Verification Index Extension (uvi)
4889 * 9.7. Location Extension (loc) (2) (3) (4)
4890 * 9.8. User Verification Method Extension (uvm)
4891 * 11. Sample scenarios
4892
4893 #authorization-gestureReferenced in:
4894 * 1.1.1. Registration
4895 * 1.1.2. Authentication
4896 * 1.1.3. Other use cases and configurations
4897 * 3. Terminology (2) (3) (4) (5) (6)
4898
4899 #biometric-recognitionReferenced in:
4900 * 3. Terminology (2)
4901
4902 #ceremonyReferenced in:
4903 * 1. Introduction
4904 * 3. Terminology (2) (3) (4) (5) (6) (7)
4905 * 6.1. Registering a new credential
4906 * 6.2. Verifying an authentication assertion
4907
4908 #clientReferenced in:
4909 * 3. Terminology
4910 * 4.1.5. Platform Authenticator Availability - PublicKeyCredential's
4911 isPlatformAuthenticatorAvailable() method (2) (3) (4)
4912
4913 #client-side-resident-credential-private-keyReferenced in:
4914 * 3. Terminology (2)
4915 * 4.1.3. Create a new credential - PublicKeyCredential's
4916 [[Create]](options) method
4917 * 4.4.3. Authenticator Selection Criteria (dictionary
4918 AuthenticatorSelectionCriteria) (2)
4919 * 5.2.1. The authenticatorMakeCredential operation
4920
4921 #conforming-user-agentReferenced in:
4922 * 1. Introduction
4923 * 2. Conformance (2) (3)
4924 * 3. Terminology (2)
4925
4926 #credential-public-keyReferenced in:

4629	* 3. Terminology
4630	* 4.2.1. Information about Public Key Credential (interface
4631	AuthenticatorAttestationResponse)
4632	* 5.1. Authenticator data
4633	* 7.4. Android Key Attestation Statement Format
4634	* 11.1. Registration (2)
4635	
4636	#credential-key-pairReferenced in:
4637	* 3. Terminology (2)
4638	* 4.1.3. Create a new credential - PublicKeyCredential's
4639	[[Create]](options) method
4640	
4641	#credential-private-keyReferenced in:
4642	* 3. Terminology (2)
4643	* 4.1. PublicKeyCredential Interface
4644	* 4.2.2. Web Authentication Assertion (interface
4645	AuthenticatorAssertionResponse)
4646	
4647	#registrationReferenced in:
4648	* 1. Introduction (2)
4649	* 3. Terminology (2) (3) (4) (5) (6) (7)
4650	* 6.1. Registering a new credential
4651	
4652	#relying-partyReferenced in:
4653	* 1. Introduction (2) (3) (4) (5)
4654	* 1.1.3. Other use cases and configurations
4655	* 3. Terminology (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13)
4656	(14)
4657	* 4. Web Authentication API (2) (3)
4658	
4659	* 4.2.1. Information about Public Key Credential (interface
4660	AuthenticatorAttestationResponse)
4661	* 4.2.2. Web Authentication Assertion (interface
4662	AuthenticatorAssertionResponse)
4663	* 4.5. Options for Credential Creation (dictionary
4664	MakeCredentialOptions) (2) (3) (4) (5)
4665	* 4.5.1. Entity Description (2) (3) (4) (5)
4666	* 4.5.2. Authenticator Selection Criteria (2)
4667	* 4.5.3. Credential Attachment enumeration (enum Attachment) (2)
	* 4.8.1. Client data used in WebAuthn signatures (dictionary
	CollectedClientData) (2) (3) (4)
4668	* 4.8.4. Credential Transport enumeration (enum ExternalTransport)
4669	(2)
4670	* 5. WebAuthn Authenticator model (2)
4671	* 5.1. Authenticator data (2)
4672	* 5.2.1. The authenticatorMakeCredential operation (2) (3) (4)
4673	* 5.2.2. The authenticatorGetAssertion operation (2) (3)
4674	* 5.3. Credential Attestation (2) (3)
4675	* 5.3.5.1. Privacy
4676	* 5.3.5.2. Attestation Certificate and Attestation Certificate CA
4677	Compromise (2) (3) (4) (5)
4678	* 6. Relying Party Operations (2) (3) (4)
4679	* 6.1. Registering a new credential (2) (3) (4) (5) (6) (7) (8) (9)
4680	(10) (11) (12) (13)
4681	* 6.2. Verifying an authentication assertion (2) (3) (4)
4682	* 7.4. Android Key Attestation Statement Format
4683	

4927	* 3. Terminology (2) (3)
4928	* 4.2.1. Information about Public Key Credential (interface
4929	AuthenticatorAttestationResponse)
4930	* 5. WebAuthn Authenticator model
4931	* 5.1. Authenticator data
4932	* 5.3. Attestation (2) (3)
4933	* 5.3.1. Attestation data (2)
4934	* 7.4. Android Key Attestation Statement Format
4935	* 11.1. Registration (2)
4936	
4937	#credential-key-pairReferenced in:
4938	* 3. Terminology (2) (3)
4939	* 4.1.3. Create a new credential - PublicKeyCredential's
4940	[[Create]](options) method
4941	
4942	#credential-private-keyReferenced in:
4943	* 3. Terminology (2) (3) (4)
4944	* 4.1. PublicKeyCredential Interface
4945	* 4.2.2. Web Authentication Assertion (interface
4946	AuthenticatorAssertionResponse)
4947	* 5. WebAuthn Authenticator model
4948	* 5.2.2. The authenticatorGetAssertion operation
4949	* 5.3. Attestation (2)
4950	
4951	#registrationReferenced in:
4952	* 1. Introduction (2)
4953	* 3. Terminology (2) (3) (4) (5) (6) (7) (8) (9)
4954	* 6.1. Registering a new credential
4955	
4956	#relying-partyReferenced in:
4957	* 1. Introduction (2) (3) (4) (5) (6) (7)
4958	* 1.1.3. Other use cases and configurations
4959	* 3. Terminology (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13)
4960	(14) (15) (16) (17) (18) (19) (20) (21) (22)
4961	* 4. Web Authentication API (2) (3) (4) (5) (6) (7)
4962	* 4.1.4. Use an existing credential to make an assertion -
4963	PublicKeyCredential's [[DiscoverFromExternalSource]](options)
4964	method (2)
4965	* 4.1.5. Platform Authenticator Availability - PublicKeyCredential's
4966	isPlatformAuthenticatorAvailable() method (2) (3)
4967	* 4.2. Authenticator Responses (interface AuthenticatorResponse)
4968	* 4.2.1. Information about Public Key Credential (interface
4969	AuthenticatorAttestationResponse) (2)
4970	* 4.2.2. Web Authentication Assertion (interface
4971	AuthenticatorAssertionResponse)
4972	* 4.4. Options for Credential Creation (dictionary
4973	MakePublicKeyCredentialOptions) (2) (3) (4) (5) (6) (7) (8)
4974	* 4.4.1. Public Key Entity Description (dictionary
4975	PublicKeyCredentialEntity) (2) (3) (4) (5)
4976	* 4.4.3. Authenticator Selection Criteria (dictionary
4977	AuthenticatorSelectionCriteria) (2) (3)
4978	* 4.4.4. Authenticator Attachment enumeration (enum
4979	AuthenticatorAttachment) (2) (3) (4)
4980	* 4.7.1. Client data used in WebAuthn signatures (dictionary
4981	CollectedClientData) (2) (3) (4)
4982	* 4.7.4. Authenticator Transport enumeration (enum
4983	AuthenticatorTransport) (2)
4984	* 5. WebAuthn Authenticator model (2)
4985	* 5.1. Authenticator data (2)
4986	* 5.2.1. The authenticatorMakeCredential operation (2) (3) (4)
4987	* 5.2.2. The authenticatorGetAssertion operation (2) (3)
4988	* 5.3. Attestation (2) (3) (4) (5) (6)
4989	* 5.3.5.1. Privacy
4990	* 5.3.5.2. Attestation Certificate and Attestation Certificate CA
4991	Compromise (2) (3) (4) (5) (6)
4992	* 6. Relying Party Operations (2) (3) (4)
4993	* 6.1. Registering a new credential (2) (3) (4) (5) (6) (7) (8) (9)
4994	(10) (11) (12) (13)
4995	* 6.2. Verifying an authentication assertion (2) (3) (4) (5)
4996	* 7.4. Android Key Attestation Statement Format

4684	* 8. WebAuthn Extensions (2) (3)
4685	* 8.2. Defining extensions (2)
4686	* 8.3. Extending request parameters (2) (3) (4)
4687	* 8.6. Example Extension (2) (3)
4688	* 9.2. Simple Transaction Authorization Extension (txAuthSimple)
4689	* 9.4. Authenticator Selection Extension (authnSel) (2)
4690	* 9.5. Supported Extensions Extension (exts) (2)
4691	* 9.6. User Verification Index Extension (uvi)
4692	* 9.7. Location Extension (loc)
4693	* 10.2. WebAuthn Extension Identifier Registrations
4694	* 11.1. Registration (2) (3) (4)
4695	* 11.2. Authentication (2) (3) (4) (5)
4696	* 11.3. Decommissioning (2)
4697	
4698	#relying-party-identifierReferenced in:
4699	* 4.5. Options for Credential Creation (dictionary
4700	MakeCredentialOptions)
4701	* 4.6. Options for Assertion Generation (dictionary
4702	PublicKeyCredentialRequestOptions)
4703	* 5. WebAuthn Authenticator model
4704	
4705	#rp-idReferenced in:
4706	* 3. Terminology (2)
4707	
4708	
4709	
4710	#public-key-credentialReferenced in:
4711	* 1. Introduction (2) (3) (4) (5)
4712	* 3. Terminology (2) (3) (4) (5) (6)
4713	* 4. Web Authentication API
4714	* 4.1. PublicKeyCredential Interface
4715	* 4.1.4. Use an existing credential -
	PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
4716	
4717	* 4.2.1. Information about Public Key Credential (interface
4718	AuthenticatorAttestationResponse)
4719	* 4.5.2. Authenticator Selection Criteria
	* 4.6. Options for Assertion Generation (dictionary
4720	PublicKeyCredentialRequestOptions)
4721	* 4.8. Supporting Data Structures
4722	* 5. WebAuthn Authenticator model (2) (3)
4723	
4724	* 5.3.3. Attestation Types
4725	
	* 5.3.5.2. Attestation Certificate and Attestation Certificate CA
	Compromise (2)

4997	* 8. WebAuthn Extensions (2) (3) (4)
4998	* 8.2. Defining extensions (2)
4999	* 8.3. Extending request parameters (2) (3) (4)
5000	* 8.6. Example Extension (2) (3)
5001	* 9.1. FIDO Appid Extension (appid) (2)
5002	* 9.2. Simple Transaction Authorization Extension (txAuthSimple)
5003	* 9.4. Authenticator Selection Extension (authnSel) (2) (3)
5004	* 9.5. Supported Extensions Extension (exts) (2)
5005	* 9.6. User Verification Index Extension (uvi)
5006	* 9.7. Location Extension (loc) (2)
5007	* 10.2. WebAuthn Extension Identifier Registrations (2)
5008	* 11.1. Registration (2) (3) (4) (5)
5009	* 11.2. Registration Specifically with Platform Authenticator (2) (3)
5010	* 11.3. Authentication (2) (3) (4) (5)
5011	* 11.4. Decommissioning (2)
5012	
5013	#relying-party-identifierReferenced in:
5014	* 4. Web Authentication API
5015	* 4.4. Options for Credential Creation (dictionary
5016	MakePublicKeyCredentialOptions)
5017	* 4.5. Options for Assertion Generation (dictionary
5018	PublicKeyCredentialRequestOptions)
5019	* 5. WebAuthn Authenticator model
5020	
5021	#rp-idReferenced in:
5022	* 3. Terminology (2) (3) (4) (5) (6)
5023	* 4. Web Authentication API (2) (3) (4) (5)
5024	* 4.1.3. Create a new credential - PublicKeyCredential's
5025	[[Create]](options) method (2)
5026	* 4.1.4. Use an existing credential to make an assertion -
5027	PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5028	method (2)
5029	* 4.4.1. Public Key Entity Description (dictionary
5030	PublicKeyCredentialEntity)
5031	* 5. WebAuthn Authenticator model
5032	* 5.1. Authenticator data (2) (3) (4) (5) (6)
5033	* 5.2.1. The authenticatorMakeCredential operation (2) (3)
5034	* 5.2.2. The authenticatorGetAssertion operation (2) (3)
5035	* 6.1. Registering a new credential (2)
5036	* 6.2. Verifying an authentication assertion (2)
5037	* 7.4. Android Key Attestation Statement Format
5038	* 7.6. FIDO U2F Attestation Statement Format (2) (3)
5039	
5040	#public-key-credentialReferenced in:
5041	* 1. Introduction (2) (3) (4) (5)
5042	* 3. Terminology (2) (3) (4) (5) (6) (7) (8)
5043	* 4. Web Authentication API (2) (3) (4)
5044	* 4.1. PublicKeyCredential Interface
5045	* 4.1.3. Create a new credential - PublicKeyCredential's
5046	[[Create]](options) method
5047	* 4.1.4. Use an existing credential to make an assertion -
5048	PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5049	method (2)
5050	* 4.2.1. Information about Public Key Credential (interface
5051	AuthenticatorAttestationResponse)
5052	* 4.4.1. Public Key Entity Description (dictionary
5053	PublicKeyCredentialEntity)
5054	* 4.4.3. Authenticator Selection Criteria (dictionary
5055	AuthenticatorSelectionCriteria)
5056	* 4.5. Options for Assertion Generation (dictionary
5057	PublicKeyCredentialRequestOptions)
5058	* 4.7. Supporting Data Structures
5059	* 5. WebAuthn Authenticator model (2) (3) (4) (5)
5060	* 5.2.2. The authenticatorGetAssertion operation (2) (3)
5061	* 5.3. Attestation (2)
5062	* 5.3.2. Attestation Statement Formats
5063	* 5.3.3. Attestation Types
5064	* 5.3.4. Generating an Attestation Object
5065	* 5.3.5.2. Attestation Certificate and Attestation Certificate CA
5066	Compromise (2)

* 6.1. Registering a new credential
 * 8. WebAuthn Extensions (2)
 * 11. Sample scenarios

#test-of-user-presenceReferenced in:
 * 3. Terminology (2) (3) (4) (5) (6)

* 9.2. Simple Transaction Authorization Extension (txAuthSimple)
 * 9.3. Generic Transaction Authorization Extension (txAuthGeneric)

#user-consentReferenced in:
 * 1. Introduction
 * 3. Terminology (2)
 * 4.1.3. Create a new credential - PublicKeyCredential's [[Create]](options) method
 * 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
 * 5. WebAuthn Authenticator model (2) (3)
 * 5.2.2. The authenticatorGetAssertion operation (2)

#user-verificationReferenced in:
 * 1. Introduction
 * 3. Terminology (2) (3) (4) (5) (6) (7) (8)
 * 4.1.3. Create a new credential - PublicKeyCredential's [[Create]](options) method
 * 9.2. Simple Transaction Authorization Extension (txAuthSimple)
 * 9.3. Generic Transaction Authorization Extension (txAuthGeneric)

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

#upReferenced in:
 * 5.1. Authenticator data

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

#uvReferenced in:
 * 5.1. Authenticator data

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

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

#publickeycredentialReferenced in:
 * 1. Introduction
 * 4.1. PublicKeyCredential Interface (2) (3) (4) (5) (6) (7) (8)
 * 4.1.3. Create a new credential - PublicKeyCredential's [[Create]](options) method (2) (3) (4) (5) (6)
 * 4.1.4. Use an existing credential to make an assertion - PublicKeyCredential's [[DiscoverFromExternalSource]](options) method (2) (3)
 * 4.1.5. Platform Authenticator Availability - PublicKeyCredential's isPlatformAuthenticatorAvailable() method
 * 4.7.3. Credential Descriptor (dictionary)

4776 PublicKeyCredentialDescriptor)
4777 * 5.2.1. The authenticatorMakeCredential operation (2)
4778 * 6. Relying Party Operations
4779 * 6.2. Verifying an authentication assertion
4780
4781 #dom-publickeycredential-rawidReferenced in:
4782 * 4.1. PublicKeyCredential Interface
4783 * 6.2. Verifying an authentication assertion
4784
4785 #dom-publickeycredential-responseReferenced in:
4786 * 4.1. PublicKeyCredential Interface
4787 * 4.1.3. Create a new credential - PublicKeyCredential's
4788 [[Create]](options) method
4789 * 4.1.4. Use an existing credential -
4790 PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
4791
4792 * 6.2. Verifying an authentication assertion
4793
4794 #dom-publickeycredential-clientextensionresultsReferenced in:
4795 * 4.1. PublicKeyCredential Interface
4796 * 4.1.3. Create a new credential - PublicKeyCredential's
4797 [[Create]](options) method
4798 * 4.1.4. Use an existing credential -
4799 PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
4800
4801 * 8.4. Client extension processing
4802
4803 #dom-publickeycredential-identifier-slotReferenced in:
4804 * 4.1. PublicKeyCredential Interface (2)
4805 * 4.1.3. Create a new credential - PublicKeyCredential's
4806 [[Create]](options) method
4807 * 4.1.4. Use an existing credential -
4808 PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
4809
4810 #dictdef-credentialrequestoptionsReferenced in:
4811 * 4.1.1. CredentialRequestOptions Extension
4812 * 4.1.4. Use an existing credential -
4813 PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
4814
4815 #dom-credentialrequestoptions-publickeyReferenced in:
4816 * 4.1.3. Create a new credential - PublicKeyCredential's
4817 [[Create]](options) method (2)
4818 * 4.1.4. Use an existing credential -
4819 PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
4820
4821 #dom-credentialcreationoptions-publickeyReferenced in:
4822 * 4.1.3. Create a new credential - PublicKeyCredential's
4823 [[Create]](options) method
4824
4825
4826 #dom-publickeycredential-create-slotReferenced in:
4827 * 4.1. PublicKeyCredential Interface
4828
4829 #dom-publickeycredential-create-options-optionsReferenced in:
4830 * 6.1. Registering a new credential
4831
4832 #dom-publickeycredential-discoverfromexternalsource-slotReferenced in:
4833 * 4.1. PublicKeyCredential Interface
4834
4835 #authenticatorresponseReferenced in:
4836 * 4.1. PublicKeyCredential Interface (2)
4837 * 4.2. Authenticator Responses (interface AuthenticatorResponse) (2)
4838 * 4.2.1. Information about Public Key Credential (interface
4839 AuthenticatorAttestationResponse) (2)
4840 * 4.2.2. Web Authentication Assertion (interface
4841 AuthenticatorAssertionResponse) (2)

5126 PublicKeyCredentialDescriptor)
5127 * 5.2.1. The authenticatorMakeCredential operation
5128 * 6. Relying Party Operations
5129 * 6.2. Verifying an authentication assertion
5130
5131 #dom-publickeycredential-rawidReferenced in:
5132 * 4.1. PublicKeyCredential Interface
5133 * 6.2. Verifying an authentication assertion
5134
5135 #dom-publickeycredential-responseReferenced in:
5136 * 4.1. PublicKeyCredential Interface
5137 * 4.1.3. Create a new credential - PublicKeyCredential's
5138 [[Create]](options) method
5139 * 4.1.4. Use an existing credential to make an assertion -
5140 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5141 method
5142 * 6.2. Verifying an authentication assertion
5143
5144 #dom-publickeycredential-clientextensionresultsReferenced in:
5145 * 4.1. PublicKeyCredential Interface
5146 * 4.1.3. Create a new credential - PublicKeyCredential's
5147 [[Create]](options) method
5148 * 4.1.4. Use an existing credential to make an assertion -
5149 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5150 method
5151 * 8.4. Client extension processing
5152
5153 #dom-publickeycredential-identifier-slotReferenced in:
5154 * 4.1. PublicKeyCredential Interface (2)
5155 * 4.1.3. Create a new credential - PublicKeyCredential's
5156 [[Create]](options) method
5157 * 4.1.4. Use an existing credential to make an assertion -
5158 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5159 method
5160
5161 #dom-credentialcreationoptions-publickeyReferenced in:
5162 * 4.1.3. Create a new credential - PublicKeyCredential's
5163 [[Create]](options) method (2) (3)
5164
5165 #dom-credentialrequestoptions-publickeyReferenced in:
5166 * 4.1.4. Use an existing credential to make an assertion -
5167 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5168 method (2) (3)
5169
5170 #dom-publickeycredential-create-slotReferenced in:
5171 * 4.1. PublicKeyCredential Interface
5172
5173 #dom-publickeycredential-create-options-optionsReferenced in:
5174 * 6.1. Registering a new credential
5175
5176 #dom-publickeycredential-discoverfromexternalsource-slotReferenced in:
5177 * 4.1. PublicKeyCredential Interface
5178
5179 #authenticatorresponseReferenced in:
5180 * 4.1. PublicKeyCredential Interface (2)
5181 * 4.2. Authenticator Responses (interface AuthenticatorResponse) (2)
5182 * 4.2.1. Information about Public Key Credential (interface
5183 AuthenticatorAttestationResponse) (2)
5184 * 4.2.2. Web Authentication Assertion (interface
5185 AuthenticatorAssertionResponse) (2)

```
4839 #dom-authenticatorresponse-clientdatajsonReferenced in:
4840 * 4.1.3. Create a new credential - PublicKeyCredential's
4841   [[Create]](options) method (2)
4842 * 4.1.4. Use an existing credential -
4843   PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
4844   (2)
4845 * 4.2. Authenticator Responses (interface AuthenticatorResponse)
4846 * 4.2.1. Information about Public Key Credential (interface
4847   AuthenticatorAttestationResponse)
4848 * 4.2.2. Web Authentication Assertion (interface
4849   AuthenticatorAssertionResponse)
4850 * 6.1. Registering a new credential (2)
4851 * 6.2. Verifying an authentication assertion
4852
4853 #authenticatorattestationresponseReferenced in:
4854 * 4.1. PublicKeyCredential Interface
4855 * 4.1.3. Create a new credential - PublicKeyCredential's
4856   [[Create]](options) method (2)
4857 * 4.2.1. Information about Public Key Credential (interface
4858   AuthenticatorAttestationResponse) (2)
4859 * 6. Relying Party Operations
4860 * 6.1. Registering a new credential (2) (3)
4861
4862 #dom-authenticatorattestationresponse-attestationobjectReferenced in:
4863 * 4.1.3. Create a new credential - PublicKeyCredential's
4864   [[Create]](options) method
4865 * 4.2.1. Information about Public Key Credential (interface
4866   AuthenticatorAttestationResponse)
4867 * 6.1. Registering a new credential
4868
4869 #authenticatorassertionresponseReferenced in:
4870 * 3. Terminology
4871 * 4.1. PublicKeyCredential Interface
4872 * 4.1.4. Use an existing credential -
4873   PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
4874   (2)
4875 * 4.2.2. Web Authentication Assertion (interface
4876   AuthenticatorAssertionResponse) (2)
4877 * 6. Relying Party Operations
4878
4879 #dom-authenticatorassertionresponse-authenticatordataReferenced in:
4880 * 4.1.4. Use an existing credential -
4881   PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
4882   (2)
4883 * 4.2.2. Web Authentication Assertion (interface
4884   AuthenticatorAssertionResponse)
4885 * 6.2. Verifying an authentication assertion
4886
4887 #dom-authenticatorassertionresponse-signatureReferenced in:
4888 * 4.1.4. Use an existing credential -
4889   PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
4890   (2)
4891 * 4.2.2. Web Authentication Assertion (interface
4892   AuthenticatorAssertionResponse)
4893 * 6.2. Verifying an authentication assertion
4894
4895 #dictdef-publickeycredentialparametersReferenced in:
4896 * 4.3. Parameters for Credential Generation (dictionary
4897   PublicKeyCredentialParameters)
4898 * 4.5. Options for Credential Creation (dictionary
4899   MakeCredentialOptions) (2)
4900
4901 #dom-publickeycredentialparameters-typeReferenced in:
4902 * 4.1.3. Create a new credential - PublicKeyCredential's
4903   [[Create]](options) method (2)
4904 * 4.3. Parameters for Credential Generation (dictionary
4905   PublicKeyCredentialParameters)
4906
4907 #dom-publickeycredentialparameters-algorithmReferenced in:
```

```
5186 #dom-authenticatorresponse-clientdatajsonReferenced in:
5187 * 4.1.3. Create a new credential - PublicKeyCredential's
5188   [[Create]](options) method (2)
5189 * 4.1.4. Use an existing credential to make an assertion -
5190   PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5191   method (2)
5192 * 4.2. Authenticator Responses (interface AuthenticatorResponse)
5193 * 4.2.1. Information about Public Key Credential (interface
5194   AuthenticatorAttestationResponse)
5195 * 4.2.2. Web Authentication Assertion (interface
5196   AuthenticatorAssertionResponse)
5197 * 6.1. Registering a new credential (2)
5198 * 6.2. Verifying an authentication assertion
5199
5200 #authenticatorattestationresponseReferenced in:
5201 * 4.1. PublicKeyCredential Interface
5202 * 4.1.3. Create a new credential - PublicKeyCredential's
5203   [[Create]](options) method (2)
5204 * 4.2.1. Information about Public Key Credential (interface
5205   AuthenticatorAttestationResponse) (2)
5206 * 6. Relying Party Operations
5207 * 6.1. Registering a new credential (2) (3)
5208
5209 #dom-authenticatorattestationresponse-attestationobjectReferenced in:
5210 * 4.1.3. Create a new credential - PublicKeyCredential's
5211   [[Create]](options) method
5212 * 4.2.1. Information about Public Key Credential (interface
5213   AuthenticatorAttestationResponse)
5214 * 6.1. Registering a new credential
5215
5216 #authenticatorassertionresponseReferenced in:
5217 * 3. Terminology
5218 * 4.1. PublicKeyCredential Interface
5219 * 4.1.4. Use an existing credential to make an assertion -
5220   PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5221   method
5222 * 4.2.2. Web Authentication Assertion (interface
5223   AuthenticatorAssertionResponse) (2)
5224 * 6. Relying Party Operations
5225
5226 #dom-authenticatorassertionresponse-authenticatordataReferenced in:
5227 * 4.1.4. Use an existing credential to make an assertion -
5228   PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5229   method (2)
5230 * 4.2.2. Web Authentication Assertion (interface
5231   AuthenticatorAssertionResponse)
5232 * 6.2. Verifying an authentication assertion
5233
5234 #dom-authenticatorassertionresponse-signatureReferenced in:
5235 * 4.1.4. Use an existing credential to make an assertion -
5236   PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5237   method (2)
5238 * 4.2.2. Web Authentication Assertion (interface
5239   AuthenticatorAssertionResponse)
5240 * 6.2. Verifying an authentication assertion
5241
5242 #dictdef-publickeycredentialparametersReferenced in:
5243 * 4.3. Parameters for Credential Generation (dictionary
5244   PublicKeyCredentialParameters)
5245 * 4.4. Options for Credential Creation (dictionary
5246   MakePublicKeyCredentialOptions) (2)
5247
5248 #dom-publickeycredentialparameters-typeReferenced in:
5249 * 4.1.3. Create a new credential - PublicKeyCredential's
5250   [[Create]](options) method (2)
5251 * 4.3. Parameters for Credential Generation (dictionary
5252   PublicKeyCredentialParameters)
5253
5254 #dom-publickeycredentialparameters-algReferenced in:
```


4908 * 4.1.3. Create a new credential - PublicKeyCredential's
4909 \[Create]](options) method
4910 * 4.3. Parameters for Credential Generation (dictionary
4911 PublicKeyCredentialParameters)
4912
4913 #dictdef-publickeycredentialuserentityReferenced in:
4914 * 4.4. User Account Parameters for Credential Generation (dictionary
4915 PublicKeyCredentialUserEntity)
4916 * 4.5. Options for Credential Creation (dictionary
4917 MakeCredentialOptions) (2)
4918
4919 #dom-publickeycredentialuserentity-displaynameReferenced in:
4920 * 4.1.3. Create a new credential - PublicKeyCredential's
4921 \[Create]](options) method
4922 * 4.4. User Account Parameters for Credential Generation (dictionary
4923 PublicKeyCredentialUserEntity)
4924 * 4.5. Options for Credential Creation (dictionary
4925 MakeCredentialOptions)
4926
4927 #dictdef-makecredentialoptionsReferenced in:
4928 * 4.1.2. CredentialCreationOptions Extension
4929 * 4.1.3. Create a new credential - PublicKeyCredential's
4930 \[Create]](options) method
4931 * 4.5. Options for Credential Creation (dictionary
4932 MakeCredentialOptions)
4933
4934 #dom-makecredentialoptions-rpReferenced in:
4935 * 4.1.3. Create a new credential - PublicKeyCredential's
4936 \[Create]](options) method (2) (3) (4) (5)
4937 * 4.5. Options for Credential Creation (dictionary
4938 MakeCredentialOptions)
4939
4940 #dom-makecredentialoptions-userReferenced in:
4941 * 4.1.3. Create a new credential - PublicKeyCredential's
4942 \[Create]](options) method (2) (3) (4)
4943 * 4.5. Options for Credential Creation (dictionary
4944 MakeCredentialOptions)
4945 * 5.2.1. The authenticatorMakeCredential operation (2)
4946 * 6.1. Registering a new credential
4947
4948 #dom-makecredentialoptions-challengeReferenced in:
4949 * 4.1.3. Create a new credential - PublicKeyCredential's
4950 \[Create]](options) method
4951 * 4.5. Options for Credential Creation (dictionary
4952 MakeCredentialOptions)
4953
4954 #dom-makecredentialoptions-parametersReferenced in:
4955 * 4.1.3. Create a new credential - PublicKeyCredential's
4956 \[Create]](options) method (2)
4957 * 4.5. Options for Credential Creation (dictionary
4958 MakeCredentialOptions)
4959
4960 #dom-makecredentialoptions-timeoutReferenced in:
4961 * 4.1.3. Create a new credential - PublicKeyCredential's
4962 \[Create]](options) method (2)
4963 * 4.5. Options for Credential Creation (dictionary
4964 MakeCredentialOptions)
4965
4966 #dom-makecredentialoptions-excludelistReferenced in:
4967 * 4.1.3. Create a new credential - PublicKeyCredential's
4968 \[Create]](options) method
4969 * 4.5. Options for Credential Creation (dictionary
4970 MakeCredentialOptions)
4971
4972 #dom-makecredentialoptions-authenticatorselectionReferenced in:
4973
4974 * 4.1.3. Create a new credential - PublicKeyCredential's
4975 \[Create]](options) method
4976 * 4.5. Options for Credential Creation (dictionary
4977 MakeCredentialOptions)

5256 * 4.1.3. Create a new credential - PublicKeyCredential's
5257 \[Create]](options) method
5258 * 4.3. Parameters for Credential Generation (dictionary
5259 PublicKeyCredentialParameters)
5260
5261 #dictdef-makepublickeycredentialoptionsReferenced in:
5262 * 4.1.1. CredentialCreationOptions Extension
5263
5264
5265
5266 * 4.1.3. Create a new credential - PublicKeyCredential's
5267 \[Create]](options) method
5268 * 4.4. Options for Credential Creation (dictionary
5269 MakePublicKeyCredentialOptions)
5270
5271 #dom-makepublickeycredentialoptions-rpReferenced in:
5272 * 4.1.3. Create a new credential - PublicKeyCredential's
5273 \[Create]](options) method (2) (3) (4) (5) (6)
5274 * 4.4. Options for Credential Creation (dictionary
5275 MakePublicKeyCredentialOptions)
5276
5277 #dom-makepublickeycredentialoptions-userReferenced in:
5278 * 4.1.3. Create a new credential - PublicKeyCredential's
5279 \[Create]](options) method (2) (3) (4)
5280 * 4.4. Options for Credential Creation (dictionary
5281 MakePublicKeyCredentialOptions)
5282 * 5.2.1. The authenticatorMakeCredential operation (2)
5283 * 6.1. Registering a new credential
5284
5285 #dom-makepublickeycredentialoptions-challengeReferenced in:
5286 * 4.1.3. Create a new credential - PublicKeyCredential's
5287 \[Create]](options) method
5288 * 4.4. Options for Credential Creation (dictionary
5289 MakePublicKeyCredentialOptions)
5290
5291 #dom-makepublickeycredentialoptions-pubkeycredparamsReferenced in:
5292 * 4.1.3. Create a new credential - PublicKeyCredential's
5293 \[Create]](options) method (2)
5294 * 4.4. Options for Credential Creation (dictionary
5295 MakePublicKeyCredentialOptions)
5296
5297 #dom-makepublickeycredentialoptions-timeoutReferenced in:
5298 * 4.1.3. Create a new credential - PublicKeyCredential's
5299 \[Create]](options) method (2)
5300 * 4.4. Options for Credential Creation (dictionary
5301 MakePublicKeyCredentialOptions)
5302
5303 #dom-makepublickeycredentialoptions-excludecredentialsReferenced in:
5304 * 4.1.3. Create a new credential - PublicKeyCredential's
5305 \[Create]](options) method
5306 * 4.4. Options for Credential Creation (dictionary
5307 MakePublicKeyCredentialOptions)
5308
5309 #dom-makepublickeycredentialoptions-authenticatorselectionReferenced
5310 in:
5311 * 4.1.3. Create a new credential - PublicKeyCredential's
5312 \[Create]](options) method (2)
5313 * 4.4. Options for Credential Creation (dictionary
5314 MakePublicKeyCredentialOptions)

```

5312 * 5.2.1. The authenticatorMakeCredential operation (2)
5313
5314 #dom-makepublickeycredentialoptions-extensionsReferenced in:
5315 * 4.1.3. Create a new credential - PublicKeyCredential's
5316 [[Create]](options) method (2)
5317 * 4.4. Options for Credential Creation (dictionary
5318 MakePublicKeyCredentialOptions)
5319 * 8.3. Extending request parameters
5320
5321 #dictdef-publickeycredentialentityReferenced in:
5322 * 4.4. Options for Credential Creation (dictionary
5323 MakePublicKeyCredentialOptions) (2)
5324 * 4.4.1. Public Key Entity Description (dictionary
5325 PublicKeyCredentialEntity) (2)
5326 * 4.4.2. User Account Parameters for Credential Generation
5327 (dictionary PublicKeyCredentialUserEntity)
5328 * 5.2.1. The authenticatorMakeCredential operation
5329
5330 #dom-publickeycredentialentity-idReferenced in:
5331 * 4.1.3. Create a new credential - PublicKeyCredential's
5332 [[Create]](options) method (2) (3) (4) (5)
5333 * 4.4. Options for Credential Creation (dictionary
5334 MakePublicKeyCredentialOptions) (2) (3)
5335 * 4.4.1. Public Key Entity Description (dictionary
5336 PublicKeyCredentialEntity)
5337 * 5.2.1. The authenticatorMakeCredential operation (2)
5338
5339 #dom-publickeycredentialentity-nameReferenced in:
5340 * 4.1.3. Create a new credential - PublicKeyCredential's
5341 [[Create]](options) method (2)
5342 * 4.4. Options for Credential Creation (dictionary
5343 MakePublicKeyCredentialOptions) (2)
5344 * 4.4.1. Public Key Entity Description (dictionary
5345 PublicKeyCredentialEntity)
5346
5347 #dom-publickeycredentialentity-iconReferenced in:
5348 * 4.4.1. Public Key Entity Description (dictionary
5349 PublicKeyCredentialEntity)
5350
5351 #dictdef-publickeycredentialuserentityReferenced in:
5352 * 4.4. Options for Credential Creation (dictionary
5353 MakePublicKeyCredentialOptions) (2)
5354 * 4.4.2. User Account Parameters for Credential Generation
5355 (dictionary PublicKeyCredentialUserEntity) (2)
5356 * 5.2.1. The authenticatorMakeCredential operation
5357
5358 #dom-publickeycredentialuserentity-displaynameReferenced in:
5359 * 4.1.3. Create a new credential - PublicKeyCredential's
5360 [[Create]](options) method
5361 * 4.4. Options for Credential Creation (dictionary
5362 MakePublicKeyCredentialOptions)
5363 * 4.4.2. User Account Parameters for Credential Generation
5364 (dictionary PublicKeyCredentialUserEntity)
5365
5366 #dictdef-authenticatorselectioncriteriaReferenced in:
5367 * 4.4. Options for Credential Creation (dictionary
5368 MakePublicKeyCredentialOptions) (2)
5369 * 4.4.3. Authenticator Selection Criteria (dictionary
5370 AuthenticatorSelectionCriteria) (2)
5371
5372 #dom-authenticatorselectioncriteria-aaReferenced in:
5373 * 4.1.3. Create a new credential - PublicKeyCredential's
5374 [[Create]](options) method
5375 * 4.4.3. Authenticator Selection Criteria (dictionary
5376 AuthenticatorSelectionCriteria)
5377
5378 #dom-authenticatorselectioncriteria-rkReferenced in:
5379 * 4.1.3. Create a new credential - PublicKeyCredential's
5380 [[Create]](options) method (2)
5381 * 4.4.3. Authenticator Selection Criteria (dictionary

```

5026
5027
5028
5029
#enumdef-attachmentReferenced in:
* 4.5.2. Authenticator Selection Criteria (2)
* 4.5.3. Credential Attachment enumeration (enum Attachment) (2)

5030
5031
5032
#platform-authenticatorsReferenced in:
* 4.5.3. Credential Attachment enumeration (enum Attachment) (2)

5033
5034
5035
5036
#roaming-authenticatorsReferenced in:
* 1.1.3. Other use cases and configurations
* 4.5.3. Credential Attachment enumeration (enum Attachment) (2)

5037
5038
5039
#platform-attachmentReferenced in:
* 4.5.3. Credential Attachment enumeration (enum Attachment)

5040
5041
5042
#cross-platform-attachedReferenced in:
* 4.5.3. Credential Attachment enumeration (enum Attachment) (2)

5043
5044
5045
5046
5047
5048
5049
#dictdef-publickeycredentialrequestoptionsReferenced in:
* 4.1.1. CredentialRequestOptions Extension
* 4.6. Options for Assertion Generation (dictionary
PublicKeyCredentialRequestOptions) (2)
* 6.2. Verifying an authentication assertion

5050
5051
5052
5053
#dom-publickeycredentialrequestoptions-challengeReferenced in:
* 4.1.4. Use an existing credential -
PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
* 4.6. Options for Assertion Generation (dictionary
PublicKeyCredentialRequestOptions) (2)

5054
5055
5056
5057
5058
5059
5060
#dom-publickeycredentialrequestoptions-timeoutReferenced in:
* 4.1.4. Use an existing credential -
PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
(2)
* 4.6. Options for Assertion Generation (dictionary
PublicKeyCredentialRequestOptions)

5061
5062
5063
5064
5065
5066
5067
#dom-publickeycredentialrequestoptions-rpidReferenced in:
* 4.1.4. Use an existing credential -
PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
(2) (3)
* 4.6. Options for Assertion Generation (dictionary
PublicKeyCredentialRequestOptions)
* 9.1. FIDO AppId Extension (appid)

5068
5069
5070
5071
5072
5073
5074
5075
5076
#dom-publickeycredentialrequestoptions-allowlistReferenced in:
* 4.1.4. Use an existing credential -
PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
(2) (3) (4)
* 4.6. Options for Assertion Generation (dictionary
PublicKeyCredentialRequestOptions)

5382
5383
5384
5385
5386
5387
5388
5389
5390
5391
5392
5393
5394
5395
5396
5397
5398
5399
5400
5401
5402
5403
5404
5405
5406
5407
5408
5409
5410
5411
5412
5413
5414
5415
5416
5417
5418
5419
5420
5421
5422
5423
5424
5425
5426
5427
5428
5429
5430
5431
5432
5433
5434
5435
5436
5437
5438
5439
5440
5441
5442
5443
5444
5445
5446
5447
5448
5449
5450
5451
AuthenticatorSelectionCriteria)

#dom-authenticatorselectioncriteria-uvReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's
[[Create]](options) method
* 4.4.3. Authenticator Selection Criteria (dictionary
AuthenticatorSelectionCriteria)

#enumdef-authenticatorattachmentReferenced in:
* 4.4.3. Authenticator Selection Criteria (dictionary
AuthenticatorSelectionCriteria) (2)
* 4.4.4. Authenticator Attachment enumeration (enum
AuthenticatorAttachment) (2)

#platform-authenticatorsReferenced in:
* 4.1.5. Platform Authenticator Availability - PublicKeyCredential's
isPlatformAuthenticatorAvailable() method (2) (3) (4) (5)
* 4.4.4. Authenticator Attachment enumeration (enum
AuthenticatorAttachment) (2)
* 11.1. Registration
* 11.2. Registration Specifically with Platform Authenticator (2)

#roaming-authenticatorsReferenced in:
* 1.1.3. Other use cases and configurations
* 4.4.4. Authenticator Attachment enumeration (enum
AuthenticatorAttachment) (2)
* 11.1. Registration

#platform-attachmentReferenced in:
* 4.4.4. Authenticator Attachment enumeration (enum
AuthenticatorAttachment)

#cross-platform-attachedReferenced in:
* 4.4.4. Authenticator Attachment enumeration (enum
AuthenticatorAttachment) (2)

#dictdef-publickeycredentialrequestoptionsReferenced in:
* 4.1.2. CredentialRequestOptions Extension
* 4.5. Options for Assertion Generation (dictionary
PublicKeyCredentialRequestOptions) (2)
* 6.2. Verifying an authentication assertion

#dom-publickeycredentialrequestoptions-challengeReferenced in:
* 4.1.4. Use an existing credential to make an assertion -
PublicKeyCredential's [[DiscoverFromExternalSource]](options)
method
* 4.5. Options for Assertion Generation (dictionary
PublicKeyCredentialRequestOptions) (2)

#dom-publickeycredentialrequestoptions-timeoutReferenced in:
* 4.1.4. Use an existing credential to make an assertion -
PublicKeyCredential's [[DiscoverFromExternalSource]](options)
method (2)
* 4.5. Options for Assertion Generation (dictionary
PublicKeyCredentialRequestOptions)

#dom-publickeycredentialrequestoptions-rpidReferenced in:
* 4.1.4. Use an existing credential to make an assertion -
PublicKeyCredential's [[DiscoverFromExternalSource]](options)
method (2) (3) (4)
* 4.5. Options for Assertion Generation (dictionary
PublicKeyCredentialRequestOptions)
* 9.1. FIDO AppId Extension (appid)

#dom-publickeycredentialrequestoptions-allowcredentialsReferenced in:
* 4.1.4. Use an existing credential to make an assertion -
PublicKeyCredential's [[DiscoverFromExternalSource]](options)
method (2) (3) (4)
* 4.5. Options for Assertion Generation (dictionary
PublicKeyCredentialRequestOptions)


```
5077 #dom-publickeycredentialrequestoptions-extensionsReferenced in:
5078 * 4.1.4. Use an existing credential -
5079   PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
5080   (2)
5081 * 4.6. Options for Assertion Generation (dictionary
5082   PublicKeyCredentialRequestOptions)
5083
5084 #typedefdef-authenticationextensionsReferenced in:
5085 * 4.1. PublicKeyCredential Interface (2)
5086 * 4.1.3. Create a new credential - PublicKeyCredential's
5087   [[Create]](options) method
5088 * 4.1.4. Use an existing credential -
5089   PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
5090 * 4.5. Options for Credential Creation (dictionary
5091   MakeCredentialOptions) (2)
5092 * 4.6. Options for Assertion Generation (dictionary
5093   PublicKeyCredentialRequestOptions) (2)
5094 * 4.8.1. Client data used in WebAuthn signatures (dictionary
5095   CollectedClientData) (2)
5096
5097 #dictdef-collectedclientdataReferenced in:
5098 * 4.1.3. Create a new credential - PublicKeyCredential's
5099   [[Create]](options) method
5100 * 4.1.4. Use an existing credential -
5101   PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
5102 * 4.8.1. Client data used in WebAuthn signatures (dictionary
5103   CollectedClientData) (2)
5104
5105 #client-dataReferenced in:
5106 * 4.2. Authenticator Responses (interface AuthenticatorResponse)
5107 * 5. WebAuthn Authenticator model (2) (3)
5108 * 5.1. Authenticator data (2)
5109 * 6.1. Registering a new credential
5110 * 6.2. Verifying an authentication assertion
5111 * 8. WebAuthn Extensions
5112 * 8.4. Client extension processing
5113 * 8.6. Example Extension
5114
5115 #dom-collectedclientdata-challengeReferenced in:
5116 * 4.1.3. Create a new credential - PublicKeyCredential's
5117   [[Create]](options) method
5118 * 4.1.4. Use an existing credential -
5119   PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
5120 * 4.8.1. Client data used in WebAuthn signatures (dictionary
5121   CollectedClientData)
5122 * 6.1. Registering a new credential
5123 * 6.2. Verifying an authentication assertion
5124
5125 #dom-collectedclientdata-originReferenced in:
5126 * 4.1.3. Create a new credential - PublicKeyCredential's
5127   [[Create]](options) method
5128 * 4.1.4. Use an existing credential -
5129   PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
5130 * 4.8.1. Client data used in WebAuthn signatures (dictionary
5131   CollectedClientData)
5132 * 6.1. Registering a new credential
5133 * 6.2. Verifying an authentication assertion
5134
5135 #dom-collectedclientdata-hashalgReferenced in:
5136 * 4.1.3. Create a new credential - PublicKeyCredential's
5137   [[Create]](options) method
5138 * 4.1.4. Use an existing credential -
5139   PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
5140 * 4.8.1. Client data used in WebAuthn signatures (dictionary
```

```
5452 #dom-publickeycredentialrequestoptions-extensionsReferenced in:
5453 * 4.1.4. Use an existing credential to make an assertion -
5454   PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5455   method (2)
5456 * 4.5. Options for Assertion Generation (dictionary
5457   PublicKeyCredentialRequestOptions)
5458
5459 #typedefdef-authenticationextensionsReferenced in:
5460 * 4.1. PublicKeyCredential Interface (2)
5461 * 4.1.3. Create a new credential - PublicKeyCredential's
5462   [[Create]](options) method
5463 * 4.1.4. Use an existing credential to make an assertion -
5464   PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5465   method
5466 * 4.4. Options for Credential Creation (dictionary
5467   MakePublicKeyCredentialOptions) (2)
5468 * 4.5. Options for Assertion Generation (dictionary
5469   PublicKeyCredentialRequestOptions) (2)
5470 * 4.7.1. Client data used in WebAuthn signatures (dictionary
5471   CollectedClientData) (2)
5472
5473 #dictdef-collectedclientdataReferenced in:
5474 * 4.1.3. Create a new credential - PublicKeyCredential's
5475   [[Create]](options) method
5476 * 4.1.4. Use an existing credential to make an assertion -
5477   PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5478   method
5479 * 4.7.1. Client data used in WebAuthn signatures (dictionary
5480   CollectedClientData) (2)
5481
5482 #client-dataReferenced in:
5483 * 4.2. Authenticator Responses (interface AuthenticatorResponse)
5484 * 5. WebAuthn Authenticator model (2) (3) (4)
5485 * 5.1. Authenticator data (2)
5486 * 6.1. Registering a new credential
5487 * 6.2. Verifying an authentication assertion
5488 * 8. WebAuthn Extensions
5489 * 8.4. Client extension processing
5490 * 8.6. Example Extension
5491
5492 #dom-collectedclientdata-challengeReferenced in:
5493 * 4.1.3. Create a new credential - PublicKeyCredential's
5494   [[Create]](options) method
5495 * 4.1.4. Use an existing credential to make an assertion -
5496   PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5497   method
5498 * 4.7.1. Client data used in WebAuthn signatures (dictionary
5499   CollectedClientData)
5500 * 6.1. Registering a new credential
5501 * 6.2. Verifying an authentication assertion
5502
5503 #dom-collectedclientdata-originReferenced in:
5504 * 4.1.3. Create a new credential - PublicKeyCredential's
5505   [[Create]](options) method
5506 * 4.1.4. Use an existing credential to make an assertion -
5507   PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5508   method
5509 * 4.7.1. Client data used in WebAuthn signatures (dictionary
5510   CollectedClientData)
5511 * 6.1. Registering a new credential
5512 * 6.2. Verifying an authentication assertion
5513
5514 #dom-collectedclientdata-hashalgorithmReferenced in:
5515 * 4.1.3. Create a new credential - PublicKeyCredential's
5516   [[Create]](options) method
5517 * 4.1.4. Use an existing credential to make an assertion -
5518   PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5519   method
5520 * 4.7.1. Client data used in WebAuthn signatures (dictionary
```

5142 CollectedClientData) (2)
5143 * 6.1. Registering a new credential
5144 * 6.2. Verifying an authentication assertion
5145
5146 #dom-collectedclientdata-tokenbindingReferenced in:
5147 * 4.1.3. Create a new credential - PublicKeyCredential's
5148 [[Create]](options) method
5149 * 4.1.4. Use an existing credential -
5150 PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
5151 * 4.8.1. Client data used in WebAuthn signatures (dictionary)
5152
5153 CollectedClientData)
5154 * 6.1. Registering a new credential
5155 * 6.2. Verifying an authentication assertion
5156
5157 #dom-collectedclientdata-clientextensionsReferenced in:
5158 * 4.1.3. Create a new credential - PublicKeyCredential's
5159 [[Create]](options) method
5160 * 4.1.4. Use an existing credential -
5161 PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
5162 * 4.8.1. Client data used in WebAuthn signatures (dictionary)
5163
5164 CollectedClientData)
5165 * 6.1. Registering a new credential
5166 * 6.2. Verifying an authentication assertion
5167 * 8.4. Client extension processing
5168
5169 #dom-collectedclientdata-authenticatorextensionsReferenced in:
5170 * 4.1.3. Create a new credential - PublicKeyCredential's
5171 [[Create]](options) method
5172 * 4.1.4. Use an existing credential -
5173 PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
5174 * 4.8.1. Client data used in WebAuthn signatures (dictionary)
5175
5176 CollectedClientData)
5177 * 6.1. Registering a new credential
5178 * 6.2. Verifying an authentication assertion
5179
5180 #collectedclientdata-json-serialized-client-dataReferenced in:
5181 * 4.1.3. Create a new credential - PublicKeyCredential's
5182 [[Create]](options) method
5183 * 4.1.4. Use an existing credential -
5184 PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
5185
5186 * 4.2. Authenticator Responses (interface AuthenticatorResponse)
5187 * 4.2.1. Information about Public Key Credential (interface
5188 AuthenticatorAttestationResponse) (2)
5189 * 4.2.2. Web Authentication Assertion (interface
5190 AuthenticatorAssertionResponse)
5191 * 4.8.1. Client data used in WebAuthn signatures (dictionary
5192 CollectedClientData)
5193
5194 #collectedclientdata-hash-of-the-serialized-client-dataReferenced in:
5195 * 4.1.3. Create a new credential - PublicKeyCredential's
5196 [[Create]](options) method (2)
5197 * 4.1.4. Use an existing credential -
5198 PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
5199 (2)
5200 * 4.2.1. Information about Public Key Credential (interface
5201 AuthenticatorAttestationResponse)
5202 * 4.2.2. Web Authentication Assertion (interface
5203 AuthenticatorAssertionResponse)
5204 * 4.8.1. Client data used in WebAuthn signatures (dictionary
5205 CollectedClientData)
5206 * 5. WebAuthn Authenticator model
5207 * 5.2.1. The authenticatorMakeCredential operation (2)
5208 * 5.2.2. The authenticatorGetAssertion operation (2) (3)
5209 * 5.3.2. Attestation Statement Formats (2)
5210 * 5.3.4. Generating an Attestation Object
5211 * 6.1. Registering a new credential

5522 CollectedClientData) (2)
5523 * 6.1. Registering a new credential
5524 * 6.2. Verifying an authentication assertion
5525
5526 #dom-collectedclientdata-tokenbindingidReferenced in:
5527 * 4.1.3. Create a new credential - PublicKeyCredential's
5528 [[Create]](options) method
5529 * 4.1.4. Use an existing credential to make an assertion -
5530 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5531 method
5532 * 4.7.1. Client data used in WebAuthn signatures (dictionary
5533 CollectedClientData)
5534 * 6.1. Registering a new credential
5535 * 6.2. Verifying an authentication assertion
5536
5537 #dom-collectedclientdata-clientextensionsReferenced in:
5538 * 4.1.3. Create a new credential - PublicKeyCredential's
5539 [[Create]](options) method
5540 * 4.1.4. Use an existing credential to make an assertion -
5541 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5542 method
5543 * 4.7.1. Client data used in WebAuthn signatures (dictionary
5544 CollectedClientData)
5545 * 6.1. Registering a new credential
5546 * 6.2. Verifying an authentication assertion
5547 * 8.4. Client extension processing
5548
5549 #dom-collectedclientdata-authenticatorextensionsReferenced in:
5550 * 4.1.3. Create a new credential - PublicKeyCredential's
5551 [[Create]](options) method
5552 * 4.1.4. Use an existing credential to make an assertion -
5553 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5554 method
5555 * 4.7.1. Client data used in WebAuthn signatures (dictionary
5556 CollectedClientData)
5557 * 6.1. Registering a new credential
5558 * 6.2. Verifying an authentication assertion
5559
5560 #collectedclientdata-json-serialized-client-dataReferenced in:
5561 * 4.1.3. Create a new credential - PublicKeyCredential's
5562 [[Create]](options) method
5563 * 4.1.4. Use an existing credential to make an assertion -
5564 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5565 method
5566 * 4.2. Authenticator Responses (interface AuthenticatorResponse)
5567 * 4.2.1. Information about Public Key Credential (interface
5568 AuthenticatorAttestationResponse) (2)
5569 * 4.2.2. Web Authentication Assertion (interface
5570 AuthenticatorAssertionResponse)
5571 * 4.7.1. Client data used in WebAuthn signatures (dictionary
5572 CollectedClientData)
5573
5574 #collectedclientdata-hash-of-the-serialized-client-dataReferenced in:
5575 * 4.1.3. Create a new credential - PublicKeyCredential's
5576 [[Create]](options) method (2)
5577 * 4.1.4. Use an existing credential to make an assertion -
5578 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5579 method (2)
5580 * 4.2.1. Information about Public Key Credential (interface
5581 AuthenticatorAttestationResponse)
5582 * 4.2.2. Web Authentication Assertion (interface
5583 AuthenticatorAssertionResponse)
5584 * 4.7.1. Client data used in WebAuthn signatures (dictionary
5585 CollectedClientData)
5586 * 5. WebAuthn Authenticator model
5587 * 5.2.1. The authenticatorMakeCredential operation (2)
5588 * 5.2.2. The authenticatorGetAssertion operation (2) (3)
5589 * 5.3.2. Attestation Statement Formats (2)
5590 * 5.3.4. Generating an Attestation Object
5591 * 6.1. Registering a new credential

```
5208 * 7.2. Packed Attestation Statement Format (2)
5209 * 7.3. TPM Attestation Statement Format (2)
5210 * 7.4. Android Key Attestation Statement Format (2)
5211 * 7.5. Android SafetyNet Attestation Statement Format
5212 * 7.6. FIDO U2F Attestation Statement Format (2)
5213
5214 #enumdef-publickeycredentialtypeReferenced in:
5215 * 4.1.3. Create a new credential - PublicKeyCredential's
5216 [[Create]](options) method (2)
5217 * 4.3. Parameters for Credential Generation (dictionary
5218 PublicKeyCredentialParameters)
5219 * 4.8.2. Credential Type enumeration (enum PublicKeyCredentialType)
5220 * 4.8.3. Credential Descriptor (dictionary
5221 PublicKeyCredentialDescriptor)
5222 * 5.2.1. The authenticatorMakeCredential operation (2) (3)
5223
5224 #dom-publickeycredentialtype-public-keyReferenced in:
5225 * 4.8.2. Credential Type enumeration (enum PublicKeyCredentialType)
5226
5227 #dictdef-publickeycredentialdescriptorReferenced in:
5228 * 4.5. Options for Credential Creation (dictionary
5229 MakeCredentialOptions) (2)
5230 * 4.6. Options for Assertion Generation (dictionary
5231 PublicKeyCredentialRequestOptions) (2) (3)
5232 * 4.8.3. Credential Descriptor (dictionary
5233 PublicKeyCredentialDescriptor)
5234
5235 #dom-publickeycredentialdescriptor-transportsReferenced in:
5236 * 4.1.3. Create a new credential - PublicKeyCredential's
5237 [[Create]](options) method (2)
5238 * 4.1.4. Use an existing credential -
5239 PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
5240 (2)
5241
5242 #dom-publickeycredentialdescriptor-typeReferenced in:
5243 * 4.1.4. Use an existing credential -
5244 PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
5245 * 4.8.3. Credential Descriptor (dictionary
5246 PublicKeyCredentialDescriptor)
5247
5248 #dom-publickeycredentialdescriptor-idReferenced in:
5249 * 4.1.4. Use an existing credential -
5250 PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
5251 * 4.8.3. Credential Descriptor (dictionary
5252 PublicKeyCredentialDescriptor)
5253
5254 #enumdef-transportReferenced in:
5255 * 4.8.3. Credential Descriptor (dictionary
5256 PublicKeyCredentialDescriptor)
5257
5258 #dom-transport-usbReferenced in:
5259 * 4.8.4. Credential Transport enumeration (enum ExternalTransport)
5260
5261 #dom-transport-nfcReferenced in:
5262 * 4.8.4. Credential Transport enumeration (enum ExternalTransport)
5263
5264 #dom-transport-bleReferenced in:
5265 * 4.8.4. Credential Transport enumeration (enum ExternalTransport)
```

```
5592 * 7.2. Packed Attestation Statement Format (2)
5593 * 7.3. TPM Attestation Statement Format (2)
5594 * 7.4. Android Key Attestation Statement Format (2)
5595 * 7.5. Android SafetyNet Attestation Statement Format
5596 * 7.6. FIDO U2F Attestation Statement Format (2)
5597
5598 #enumdef-publickeycredentialtypeReferenced in:
5599 * 4.1.3. Create a new credential - PublicKeyCredential's
5600 [[Create]](options) method (2)
5601 * 4.3. Parameters for Credential Generation (dictionary
5602 PublicKeyCredentialParameters)
5603 * 4.7.2. Credential Type enumeration (enum PublicKeyCredentialType)
5604 * 4.7.3. Credential Descriptor (dictionary
5605 PublicKeyCredentialDescriptor)
5606 * 5.2.1. The authenticatorMakeCredential operation (2) (3)
5607
5608 #dom-publickeycredentialtype-public-keyReferenced in:
5609 * 4.7.2. Credential Type enumeration (enum PublicKeyCredentialType)
5610
5611 #dictdef-publickeycredentialdescriptorReferenced in:
5612 * 4.4. Options for Credential Creation (dictionary
5613 MakePublicKeyCredentialOptions) (2)
5614 * 4.5. Options for Assertion Generation (dictionary
5615 PublicKeyCredentialRequestOptions) (2) (3)
5616 * 4.7.3. Credential Descriptor (dictionary
5617 PublicKeyCredentialDescriptor)
5618 * 5.2.1. The authenticatorMakeCredential operation
5619
5620 #dom-publickeycredentialdescriptor-transportsReferenced in:
5621 * 4.1.3. Create a new credential - PublicKeyCredential's
5622 [[Create]](options) method (2)
5623 * 4.1.4. Use an existing credential to make an assertion -
5624 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5625 method (2)
5626
5627 #dom-publickeycredentialdescriptor-typeReferenced in:
5628 * 4.1.4. Use an existing credential to make an assertion -
5629 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5630 method
5631 * 4.7.3. Credential Descriptor (dictionary
5632 PublicKeyCredentialDescriptor)
5633
5634 #dom-publickeycredentialdescriptor-idReferenced in:
5635 * 4.1.4. Use an existing credential to make an assertion -
5636 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5637 method
5638 * 4.7.3. Credential Descriptor (dictionary
5639 PublicKeyCredentialDescriptor)
5640
5641 #enumdef-authenticatortransportReferenced in:
5642 * 4.7.3. Credential Descriptor (dictionary
5643 PublicKeyCredentialDescriptor)
5644 * 4.7.4. Authenticator Transport enumeration (enum
5645 AuthenticatorTransport)
5646
5647 #dom-authenticatortransport-usbReferenced in:
5648 * 4.7.4. Authenticator Transport enumeration (enum
5649 AuthenticatorTransport)
5650
5651 #dom-authenticatortransport-nfcReferenced in:
5652 * 4.7.4. Authenticator Transport enumeration (enum
5653 AuthenticatorTransport)
5654
5655 #dom-authenticatortransport-bleReferenced in:
5656 * 4.7.4. Authenticator Transport enumeration (enum
5657 AuthenticatorTransport)
5658
5659 #typedefdef-cosealgorithmidentifierReferenced in:
5660 * 4.1.3. Create a new credential - PublicKeyCredential's
5661 [[Create]](options) method
```


5266
5267
5268
5269
5270
5271
5272
5273
5274
5275
5276
5277
5278
5279
5280
5281
5282
5283
5284
5285
5286
5287
5288
5289
5290
5291
5292

5293
5294
5295
5296
5297
5298
5299
5300
5301
5302
5303
5304
5305
5306
5307
5308
5309
5310
5311
5312
5313
5314
5315

#authenticator-dataReferenced in:
* 4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse) (2)
* 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
* 5. WebAuthn Authenticator model (2)
* 5.1. Authenticator data (2) (3) (4) (5) (6) (7) (8)
* 5.2.1. The authenticatorMakeCredential operation (2)
* 5.2.2. The authenticatorGetAssertion operation (2) (3) (4)
* 5.3. Credential Attestation (2)
* 5.3.1. Attestation data
* 5.3.2. Attestation Statement Formats (2)
* 5.3.4. Generating an Attestation Object (2) (3)
* 5.3.5.3. Attestation Certificate Hierarchy
* 6.1. Registering a new credential (2)
* 7.5. Android SafetyNet Attestation Statement Format
* 8.5. Authenticator extension processing (2)
* 8.6. Example Extension (2)
* 9.6. User Verification Index Extension (uvi)
* 9.7. Location Extension (loc)
* 9.8. User Verification Method Extension (uvm)

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

* 5.2.3. The authenticatorCancel operation (2)
* 8. WebAuthn Extensions
* 8.2. Defining extensions

#authenticatorgetassertionReferenced in:
* 3. Terminology (2) (3)
* 4.1.4. Use an existing credential - PublicKeyCredential's `[[DiscoverFromExternalSource]](options)` `method` (2) (3)
* 5. WebAuthn Authenticator model
* 5.1. Authenticator data
* 5.2.3. The authenticatorCancel operation (2)
* 8. WebAuthn Extensions
* 8.2. Defining extensions

#authenticatorcancelReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's `[[Create]](options)` method (2) (3)
* 4.1.4. Use an existing credential - PublicKeyCredential's `[[DiscoverFromExternalSource]](options)` `method` (2) (3)

#attestation-statement-formatReferenced in:

5662
5663
5664
5665
5666
5667
5668
5669
5670
5671
5672
5673
5674
5675
5676
5677
5678
5679
5680
5681
5682
5683
5684
5685
5686
5687
5688
5689
5690
5691
5692
5693
5694
5695
5696
5697
5698
5699
5700
5701
5702
5703
5704
5705
5706
5707
5708
5709
5710
5711
5712
5713
5714
5715
5716
5717
5718
5719
5720
5721
5722
5723
5724
5725
5726
5727
5728
5729
5730
5731

* 4.3. Parameters for Credential Generation (dictionary PublicKeyCredentialParameters)
* 4.7.5. Cryptographic Algorithm Identifier (typedef COSEAlgorithmIdentifier)
* 5.2.1. The authenticatorMakeCredential operation
* 5.3.1. Attestation data

#attestation-signatureReferenced in:
* 3. Terminology
* 5. WebAuthn Authenticator model (2) (3)
* 5.3. Attestation
* 7.6. FIDO U2F Attestation Statement Format

#assertion-signatureReferenced in:
* 5. WebAuthn Authenticator model (2)
* 5.2.2. The authenticatorGetAssertion operation (2) (3) (4) (5) (6)

#authenticator-dataReferenced in:
* 4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse) (2)
* 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
* 5. WebAuthn Authenticator model (2)
* 5.1. Authenticator data (2) (3) (4) (5) (6) (7) (8)
* 5.2.1. The authenticatorMakeCredential operation (2)
* 5.2.2. The authenticatorGetAssertion operation (2) (3) (4)
* 5.3. Attestation (2)
* 5.3.1. Attestation data
* 5.3.2. Attestation Statement Formats (2)
* 5.3.4. Generating an Attestation Object (2) (3)
* 5.3.5.3. Attestation Certificate Hierarchy
* 6.1. Registering a new credential (2)
* 7.5. Android SafetyNet Attestation Statement Format
* 8.5. Authenticator extension processing (2)
* 8.6. Example Extension (2)
* 9.6. User Verification Index Extension (uvi)
* 9.7. Location Extension (loc)
* 9.8. User Verification Method Extension (uvm)

#authenticatormakecredentialReferenced in:
* 3. Terminology (2) (3)
* 4.1.3. Create a new credential - PublicKeyCredential's `[[Create]](options)` method (2)
* 5. WebAuthn Authenticator model
* 5.2.3. The authenticatorCancel operation (2)
* 8. WebAuthn Extensions
* 8.2. Defining extensions

#authenticatorgetassertionReferenced in:
* 3. Terminology (2) (3)
* 4.1.4. Use an existing credential `to make an assertion` - PublicKeyCredential's `[[DiscoverFromExternalSource]](options)` `method` (2) (3) (4)
* 5. WebAuthn Authenticator model
* 5.1. Authenticator data
* 5.2.3. The authenticatorCancel operation (2)
* 8. WebAuthn Extensions
* 8.2. Defining extensions

#authenticatorcancelReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's `[[Create]](options)` method (2) (3)
* 4.1.4. Use an existing credential `to make an assertion` - PublicKeyCredential's `[[DiscoverFromExternalSource]](options)` `method` (2) (3)

#attestation-objectReferenced in:
* 3. Terminology
* 4. Web Authentication API
* 4.2.1. Information about Public Key Credential (interface

5316 * 3. Terminology
5317 * 4.2.1. Information about Public Key Credential (interface

5318 AuthenticatorAttestationResponse)

5319 * 5.3.4. Generating an Attestation Object (2)

5320 #attestation-typeReferenced in:
5321 * 3. Terminology

5322
5323 #attestation-dataReferenced in:
5324 * 5.1. Authenticator data (2) (3) (4) (5) (6) (7)
5325 * 5.2.1. The authenticatorMakeCredential operation
5326 * 5.2.2. The authenticatorGetAssertion operation
5327 * 5.3. Credential Attestation (2)
5328 * 5.3.3. Attestation Types
5329 * 6.1. Registering a new credential (2)
5330 * 7.3. TPM Attestation Statement Format
5331 * 7.4. Android Key Attestation Statement Format
5332
5333

5334 #authenticator-data-for-the-attestationReferenced in:
5335 * 7.2. Packed Attestation Statement Format
5336 * 7.3. TPM Attestation Statement Format
5337 * 7.4. Android Key Attestation Statement Format (2)
5338 * 7.5. Android SafetyNet Attestation Statement Format
5339 * 7.6. FIDO U2F Attestation Statement Format
5340

5341 #authenticator-data-claimed-to-have-been-used-for-the-attestationReferenced in:
5342 * 7.2. Packed Attestation Statement Format
5343 * 7.3. TPM Attestation Statement Format
5344 * 7.4. Android Key Attestation Statement Format (2)
5345 * 7.6. FIDO U2F Attestation Statement Format
5346
5347

5348 #basic-attestationReferenced in:
5349 * 5.3.5.1. Privacy

5350 #self-attestationReferenced in:
5351 * 3. Terminology (2) (3) (4)
5352 * 5.3. Credential Attestation
5353

5354 * 5.3.5.2. Attestation Certificate and Attestation Certificate CA
5355 Compromise

5356

5732 AuthenticatorAttestationResponse) (2)
5733 * 4.4. Options for Credential Creation (dictionary
5734 MakePublicKeyCredentialOptions) (2)
5735 * 5.2.1. The authenticatorMakeCredential operation (2)
5736 * 5.3. Attestation (2) (3)
5737 * 5.3.1. Attestation data
5738 * 5.3.4. Generating an Attestation Object (2) (3) (4)
5739 * 6.1. Registering a new credential
5740

5741 #attestation-statementReferenced in:
5742 * 3. Terminology
5743 * 4.2.1. Information about Public Key Credential (interface
5744 AuthenticatorAttestationResponse) (2) (3)
5745 * 5.3. Attestation (2) (3) (4) (5) (6) (7) (8)
5746 * 5.3.2. Attestation Statement Formats (2) (3)
5747

5748 #attestation-statement-formatReferenced in:
5749 * 4.2.1. Information about Public Key Credential (interface
5750 AuthenticatorAttestationResponse)
5751 * 4.7.4. Authenticator Transport enumeration (enum
5752 AuthenticatorTransport)
5753 * 5.3. Attestation (2) (3) (4) (5) (6) (7)
5754 * 5.3.2. Attestation Statement Formats (2) (3) (4)
5755 * 5.3.4. Generating an Attestation Object (2)
5756

5757 #attestation-typeReferenced in:
5758 * 5.3. Attestation (2) (3) (4) (5) (6)
5759 * 5.3.2. Attestation Statement Formats
5760

5761 #attestation-dataReferenced in:
5762 * 5.1. Authenticator data (2) (3) (4) (5) (6) (7)
5763 * 5.2.1. The authenticatorMakeCredential operation
5764 * 5.2.2. The authenticatorGetAssertion operation
5765 * 5.3. Attestation (2)
5766 * 5.3.3. Attestation Types
5767 * 6.1. Registering a new credential (2)
5768 * 7.3. TPM Attestation Statement Format
5769 * 7.4. Android Key Attestation Statement Format
5770

5771 #signing-procedureReferenced in:
5772 * 5.3.2. Attestation Statement Formats
5773

5774 #authenticator-data-for-the-attestationReferenced in:
5775 * 7.2. Packed Attestation Statement Format
5776 * 7.3. TPM Attestation Statement Format
5777 * 7.4. Android Key Attestation Statement Format (2)
5778 * 7.5. Android SafetyNet Attestation Statement Format
5779 * 7.6. FIDO U2F Attestation Statement Format
5780

5781 #authenticator-data-claimed-to-have-been-used-for-the-attestationReferenced in:
5782 * 7.2. Packed Attestation Statement Format
5783 * 7.3. TPM Attestation Statement Format
5784 * 7.4. Android Key Attestation Statement Format (2)
5785 * 7.6. FIDO U2F Attestation Statement Format
5786
5787

5788 #basic-attestationReferenced in:
5789 * 5.3.5.1. Privacy
5790

5791 #self-attestationReferenced in:
5792 * 3. Terminology (2) (3) (4)
5793 * 5.3. Attestation (2)
5794 * 5.3.2. Attestation Statement Formats
5795 * 5.3.3. Attestation Types
5796 * 5.3.5.2. Attestation Certificate and Attestation Certificate CA
5797 Compromise
5798 * 6.1. Registering a new credential (2) (3)
5799 * 7.2. Packed Attestation Statement Format (2)
5800 * 7.6. FIDO U2F Attestation Statement Format
5801

```
5357 #privacy-caReferenced in:
5358 * 5.3.5.1. Privacy
5359
5360 #elliptic-curve-based-direct-anonymous-attestationReferenced in:
5361 * 5.3.5.1. Privacy
5362
5363 #ecdaaReferenced in:
5364 * 5.3.2. Attestation Statement Formats
5365 * 5.3.3. Attestation Types
5366 * 5.3.5.2. Attestation Certificate and Attestation Certificate CA
5367   Compromise
5368 * 6.1. Registering a new credential
5369 * 7.2. Packed Attestation Statement Format (2)
5370 * 7.3. TPM Attestation Statement Format (2)
5371
5372 #attestation-statement-format-identifierReferenced in:
5373 * 5.3.2. Attestation Statement Formats
5374 * 5.3.4. Generating an Attestation Object
5375
5376 #identifier-of-the-ecdaa-issuer-public-keyReferenced in:
5377 * 6.1. Registering a new credential
5378 * 7.2. Packed Attestation Statement Format
5379 * 7.3. TPM Attestation Statement Format (2)
5380
5381 #ecdaa-issuer-public-keyReferenced in:
5382 * 5.3.2. Attestation Statement Formats
5383 * 5.3.5.1. Privacy
5384 * 6.1. Registering a new credential
5385 * 7.2. Packed Attestation Statement Format (2) (3)
5386
5387 #registration-extensionReferenced in:
5388 * 4.1.3. Create a new credential - PublicKeyCredential's
5389   [[Create]](options) method
5390 * 8. WebAuthn Extensions (2) (3) (4) (5) (6)
5391 * 8.6. Example Extension
5392 * 9.2. Simple Transaction Authorization Extension (txAuthSimple)
5393 * 9.3. Generic Transaction Authorization Extension (txAuthGeneric)
5394 * 9.4. Authenticator Selection Extension (authnSel)
5395 * 9.5. Supported Extensions Extension (exts)
5396 * 9.6. User Verification Index Extension (uvi)
5397 * 9.7. Location Extension (loc)
5398 * 9.8. User Verification Method Extension (uvm)
5399 * 10.2. WebAuthn Extension Identifier Registrations (2) (3) (4) (5)
5400   (6) (7)
5401
5402 #authentication-extensionReferenced in:
5403 * 4.1.4. Use an existing credential -
5404   PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
5405
5406 * 8. WebAuthn Extensions (2) (3) (4) (5) (6)
5407 * 8.6. Example Extension
5408 * 9.1. FIDO Appid Extension (appid)
5409 * 9.2. Simple Transaction Authorization Extension (txAuthSimple)
5410 * 9.3. Generic Transaction Authorization Extension (txAuthGeneric)
5411 * 9.6. User Verification Index Extension (uvi)
5412 * 9.7. Location Extension (loc)
5413 * 9.8. User Verification Method Extension (uvm)
5414 * 10.2. WebAuthn Extension Identifier Registrations (2) (3) (4) (5)
5415   (6)
5416
5417 #client-extensionReferenced in:
5418 * 4.1.3. Create a new credential - PublicKeyCredential's
5419   [[Create]](options) method
5420 * 4.1.4. Use an existing credential -
5421   PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
5422 * 4.7. Authentication Extensions (typedef AuthenticationExtensions)
5423
5424 * 8. WebAuthn Extensions
5425 * 8.2. Defining extensions
5426 * 8.4. Client extension processing
```

```
5802 #privacy-caReferenced in:
5803 * 5.3.5.1. Privacy
5804
5805 #elliptic-curve-based-direct-anonymous-attestationReferenced in:
5806 * 5.3.5.1. Privacy
5807
5808 #ecdaaReferenced in:
5809 * 5.3.2. Attestation Statement Formats
5810 * 5.3.3. Attestation Types
5811 * 5.3.5.2. Attestation Certificate and Attestation Certificate CA
5812   Compromise
5813 * 6.1. Registering a new credential
5814 * 7.2. Packed Attestation Statement Format (2)
5815 * 7.3. TPM Attestation Statement Format (2)
5816
5817 #attestation-statement-format-identifierReferenced in:
5818 * 5.3.2. Attestation Statement Formats
5819 * 5.3.4. Generating an Attestation Object
5820
5821 #identifier-of-the-ecdaa-issuer-public-keyReferenced in:
5822 * 6.1. Registering a new credential
5823 * 7.2. Packed Attestation Statement Format
5824 * 7.3. TPM Attestation Statement Format (2)
5825
5826 #ecdaa-issuer-public-keyReferenced in:
5827 * 5.3.2. Attestation Statement Formats
5828 * 5.3.5.1. Privacy
5829 * 6.1. Registering a new credential
5830 * 7.2. Packed Attestation Statement Format (2) (3)
5831
5832 #registration-extensionReferenced in:
5833 * 4.1.3. Create a new credential - PublicKeyCredential's
5834   [[Create]](options) method
5835 * 8. WebAuthn Extensions (2) (3) (4) (5) (6)
5836 * 8.6. Example Extension
5837 * 9.2. Simple Transaction Authorization Extension (txAuthSimple)
5838 * 9.3. Generic Transaction Authorization Extension (txAuthGeneric)
5839 * 9.4. Authenticator Selection Extension (authnSel)
5840 * 9.5. Supported Extensions Extension (exts)
5841 * 9.6. User Verification Index Extension (uvi)
5842 * 9.7. Location Extension (loc)
5843 * 9.8. User Verification Method Extension (uvm)
5844 * 10.2. WebAuthn Extension Identifier Registrations (2) (3) (4) (5)
5845   (6) (7)
5846
5847 #authentication-extensionReferenced in:
5848 * 4.1.4. Use an existing credential to make an assertion -
5849   PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5850   method
5851
5852 * 8. WebAuthn Extensions (2) (3) (4) (5) (6)
5853 * 8.6. Example Extension
5854 * 9.1. FIDO Appid Extension (appid)
5855 * 9.2. Simple Transaction Authorization Extension (txAuthSimple)
5856 * 9.3. Generic Transaction Authorization Extension (txAuthGeneric)
5857 * 9.6. User Verification Index Extension (uvi)
5858 * 9.7. Location Extension (loc)
5859 * 9.8. User Verification Method Extension (uvm)
5860 * 10.2. WebAuthn Extension Identifier Registrations (2) (3) (4) (5)
5861   (6)
5862
5863 #client-extensionReferenced in:
5864 * 4.1.3. Create a new credential - PublicKeyCredential's
5865   [[Create]](options) method
5866 * 4.1.4. Use an existing credential to make an assertion -
5867   PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5868   method
5869 * 4.6. Authentication Extensions (typedef AuthenticationExtensions)
5870
5871 * 8. WebAuthn Extensions
5872 * 8.2. Defining extensions
5873 * 8.4. Client extension processing
```



```
5425 #authenticator-extensionReferenced in:
5426 * 4.1.3. Create a new credential - PublicKeyCredential's
5427   [[Create]](options) method
5428 * 4.1.4. Use an existing credential -
5429   PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
5430 * 4.7. Authentication Extensions (typedef AuthenticationExtensions)
5431
5432 * 8. WebAuthn Extensions (2) (3)
5433 * 8.2. Defining extensions (2)
5434 * 8.3. Extending request parameters
5435 * 8.5. Authenticator extension processing
5436
5437 #extension-identifierReferenced in:
5438 * 4.1. PublicKeyCredential Interface
5439 * 4.1.3. Create a new credential - PublicKeyCredential's
5440   [[Create]](options) method
5441 * 4.1.4. Use an existing credential -
5442   PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
5443
5444 * 5.1. Authenticator data
5445 * 8. WebAuthn Extensions (2)
5446 * 8.2. Defining extensions
5447 * 8.3. Extending request parameters
5448 * 8.4. Client extension processing (2)
5449 * 8.5. Authenticator extension processing (2)
5450 * 8.6. Example Extension
5451 * 9.5. Supported Extensions Extension (exts) (2)
5452 * 9.7. Location Extension (loc)
5453 * 10.2. WebAuthn Extension Identifier Registrations
5454
5455 #client-extension-inputReferenced in:
5456 * 8. WebAuthn Extensions (2) (3)
5457 * 8.2. Defining extensions
5458 * 8.3. Extending request parameters (2) (3) (4) (5) (6)
5459 * 8.4. Client extension processing (2) (3) (4)
5460 * 8.6. Example Extension
5461
5462 #authenticator-extension-inputReferenced in:
5463 * 8. WebAuthn Extensions (2) (3) (4) (5)
5464 * 8.2. Defining extensions
5465 * 8.3. Extending request parameters (2) (3)
5466 * 8.4. Client extension processing
5467 * 8.5. Authenticator extension processing (2) (3)
5468
5469 #client-extension-processingReferenced in:
5470 * 4.1. PublicKeyCredential Interface
5471 * 4.1.3. Create a new credential - PublicKeyCredential's
5472   [[Create]](options) method (2)
5473 * 4.1.4. Use an existing credential -
5474   PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
5475   (2)
5476 * 8. WebAuthn Extensions (2) (3) (4)
5477 * 8.2. Defining extensions
5478
5479 #client-extension-outputReferenced in:
5480 * 4.1. PublicKeyCredential Interface
5481 * 4.1.3. Create a new credential - PublicKeyCredential's
5482   [[Create]](options) method (2)
5483 * 4.1.4. Use an existing credential -
5484   PublicKeyCredential::[[DiscoverFromExternalSource]](options) method
5485   (2)
5486 * 8. WebAuthn Extensions (2) (3)
5487 * 8.2. Defining extensions (2) (3)
5488 * 8.4. Client extension processing (2) (3)
5489 * 8.6. Example Extension
5490
5491 #authenticator-extension-processingReferenced in:
5492 * 8. WebAuthn Extensions
5493 * 8.2. Defining extensions
```

```
5872 #authenticator-extensionReferenced in:
5873 * 4.1.3. Create a new credential - PublicKeyCredential's
5874   [[Create]](options) method
5875 * 4.1.4. Use an existing credential to make an assertion -
5876   PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5877   method
5878 * 4.6. Authentication Extensions (typedef AuthenticationExtensions)
5879 * 8. WebAuthn Extensions (2) (3)
5880 * 8.2. Defining extensions (2)
5881 * 8.3. Extending request parameters
5882 * 8.5. Authenticator extension processing
5883
5884 #extension-identifierReferenced in:
5885 * 4.1. PublicKeyCredential Interface
5886 * 4.1.3. Create a new credential - PublicKeyCredential's
5887   [[Create]](options) method
5888 * 4.1.4. Use an existing credential to make an assertion -
5889   PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5890   method
5891 * 5.1. Authenticator data
5892 * 8. WebAuthn Extensions (2)
5893 * 8.2. Defining extensions
5894 * 8.3. Extending request parameters
5895 * 8.4. Client extension processing (2)
5896 * 8.5. Authenticator extension processing (2)
5897 * 8.6. Example Extension
5898 * 9.5. Supported Extensions Extension (exts) (2)
5899 * 9.7. Location Extension (loc)
5900 * 10.2. WebAuthn Extension Identifier Registrations
5901
5902 #client-extension-inputReferenced in:
5903 * 8. WebAuthn Extensions (2) (3)
5904 * 8.2. Defining extensions
5905 * 8.3. Extending request parameters (2) (3) (4) (5) (6)
5906 * 8.4. Client extension processing (2) (3) (4)
5907 * 8.6. Example Extension
5908
5909 #authenticator-extension-inputReferenced in:
5910 * 8. WebAuthn Extensions (2) (3) (4) (5)
5911 * 8.2. Defining extensions
5912 * 8.3. Extending request parameters (2) (3)
5913 * 8.4. Client extension processing
5914 * 8.5. Authenticator extension processing (2) (3)
5915
5916 #client-extension-processingReferenced in:
5917 * 4.1. PublicKeyCredential Interface
5918 * 4.1.3. Create a new credential - PublicKeyCredential's
5919   [[Create]](options) method (2)
5920 * 4.1.4. Use an existing credential to make an assertion -
5921   PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5922   method (2)
5923 * 8. WebAuthn Extensions (2) (3) (4)
5924 * 8.2. Defining extensions
5925
5926 #client-extension-outputReferenced in:
5927 * 4.1. PublicKeyCredential Interface
5928 * 4.1.3. Create a new credential - PublicKeyCredential's
5929   [[Create]](options) method (2)
5930 * 4.1.4. Use an existing credential to make an assertion -
5931   PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5932   method (2)
5933 * 8. WebAuthn Extensions (2) (3)
5934 * 8.2. Defining extensions (2) (3)
5935 * 8.4. Client extension processing (2) (3)
5936 * 8.6. Example Extension
5937
5938 #authenticator-extension-processingReferenced in:
5939 * 8. WebAuthn Extensions
5940 * 8.2. Defining extensions
```

5493 * 8.5. Authenticator extension processing
5494
5495 #authenticator-extension-outputReferenced in:
5496 * 5.1. Authenticator data
5497 * 8. WebAuthn Extensions (2) (3)
5498 * 8.2. Defining extensions (2) (3)
5499 * 8.4. Client extension processing
5500 * 8.5. Authenticator extension processing
5501 * 8.6. Example Extension
5502 * 9.5. Supported Extensions Extension (exts)
5503 * 9.6. User Verification Index Extension (uvi)
5504 * 9.7. Location Extension (loc)
5505 * 9.8. User Verification Method Extension (uvm)
5506
5507 #typedefdef-authenticatorselectionlistReferenced in:
5508 * 9.4. Authenticator Selection Extension (authnSel)
5509
5510 #typedefdef-aaguidReferenced in:
5511 * 9.4. Authenticator Selection Extension (authnSel)
5512

5942 * 8.5. Authenticator extension processing
5943
5944 #authenticator-extension-outputReferenced in:
5945 * 5.1. Authenticator data
5946 * 8. WebAuthn Extensions (2) (3)
5947 * 8.2. Defining extensions (2) (3)
5948 * 8.4. Client extension processing
5949 * 8.5. Authenticator extension processing
5950 * 8.6. Example Extension
5951 * 9.5. Supported Extensions Extension (exts)
5952 * 9.6. User Verification Index Extension (uvi)
5953 * 9.7. Location Extension (loc)
5954 * 9.8. User Verification Method Extension (uvm)
5955
5956 #typedefdef-authenticatorselectionlistReferenced in:
5957 * 9.4. Authenticator Selection Extension (authnSel)
5958
5959 #typedefdef-aaguidReferenced in:
5960 * 9.4. Authenticator Selection Extension (authnSel)
5961