

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 W3C

0005 Web Authentication: An API for accessing Public Key Credentials Level 1

0006 W3C Working Draft, 11 August 2017

0007 This version:

0008 <https://www.w3.org/TR/2017/WD-webauthn-20170811/>

0009 Latest published version:

0010 <https://www.w3.org/TR/webauthn/>

0011 Editor's Draft:

0012 <https://w3c.github.io/webauthn/>

0013 Previous Versions:

- 0014 <https://www.w3.org/TR/2017/WD-webauthn-20170505/>
- 0015 <https://www.w3.org/TR/2017/WD-webauthn-20170216/>
- 0016 <https://www.w3.org/TR/2016/WD-webauthn-20161207/>
- 0017 <https://www.w3.org/TR/2016/WD-webauthn-20160928/>
- 0018 <https://www.w3.org/TR/2016/WD-webauthn-20160902/>
- 0019 <https://www.w3.org/TR/2016/WD-webauthn-20160531/>

0020 Issue Tracking:

0021 Github

0022 Editors:

- 0023 Vijay Bharadwaj (Microsoft)
- 0024 Hubert Le Van Gong (PayPal)
- 0025 Dirk Balfanz (Google)
- 0026 Alexei Czeskis (Google)
- 0027 Arnar Birgisson (Google)
- 0028 Jeff Hodges (PayPal)
- 0029 Michael B. Jones (Microsoft)
- 0030 Rolf Lindemann (Nok Nok Labs)
- 0031 J.C. Jones (Mozilla)

0032 Tests:

0033 web-platform-tests webauthn/ (ongoing work)

0034 Copyright 2017 W3C^ (MIT, ERCIM, Keio, Beihang). W3C liability, trademark and document use rules apply.

0035 Abstract

0036 This specification defines an API enabling the creation and use of strong, attested, scoped, public key-based credentials by web applications, for the purpose of strongly authenticating users. Conceptually, one or more public key credentials, each scoped to a given Relying Party, are created and stored on an authenticator by the user agent in conjunction with the web application. The user agent mediates access to public key credentials in order to preserve user privacy. Authenticators are responsible for ensuring that no operation is performed without user consent. Authenticators provide cryptographic proof of their properties to relying parties via attestation. This specification also describes the functional model for WebAuthn conformant authenticators, including their signature and attestation functionality.

0037 Status of this document

0038

0001 THE\_URL:file://localhost/Users/jehodges/documents/work/standards/W3C/WebAuthn/index-master-121c703.html

0002 THE\_TITLE:Web Authentication: An API for accessing Public Key Credentials - Level 1

0003 W3C

0004 Web Authentication:  
0005 An API for accessing Public Key Credentials  
0006 Level 1

0007 Editor's Draft, 12 October 2017

0008 This version:

0009 <https://w3c.github.io/webauthn/>

0010 Latest published version:

0011 <https://www.w3.org/TR/webauthn/>

0012 Previous Versions:

- 0013 <https://www.w3.org/TR/2017/WD-webauthn-20170811/>
- 0014 <https://www.w3.org/TR/2017/WD-webauthn-20170505/>
- 0015 <https://www.w3.org/TR/2017/WD-webauthn-20170216/>
- 0016 <https://www.w3.org/TR/2016/WD-webauthn-20161207/>
- 0017 <https://www.w3.org/TR/2016/WD-webauthn-20160928/>
- 0018 <https://www.w3.org/TR/2016/WD-webauthn-20160902/>
- 0019 <https://www.w3.org/TR/2016/WD-webauthn-20160531/>

0020 Issue Tracking:

0021 Github

0022 Editors:

- 0023 Vijay Bharadwaj (Microsoft)
- 0024 Hubert Le Van Gong (PayPal)
- 0025 Dirk Balfanz (Google)
- 0026 Alexei Czeskis (Google)
- 0027 Arnar Birgisson (Google)
- 0028 Jeff Hodges (PayPal)
- 0029 Michael B. Jones (Microsoft)
- 0030 Rolf Lindemann (Nok Nok Labs)
- 0031 J.C. Jones (Mozilla)

0032 Tests:

0033 web-platform-tests webauthn/ (ongoing work)

0034 Copyright 2017 W3C^ (MIT, ERCIM, Keio, Beihang). W3C liability, trademark and document use rules apply.

0035 Abstract

0036 This specification defines an API enabling the creation and use of strong, attested, scoped, public key-based credentials by web applications, for the purpose of strongly authenticating users. Conceptually, one or more public key credentials, each scoped to a given Relying Party, are created and stored on an authenticator by the user agent in conjunction with the web application. The user agent mediates access to public key credentials in order to preserve user privacy. Authenticators are responsible for ensuring that no operation is performed without user consent. Authenticators provide cryptographic proof of their properties to relying parties via attestation. This specification also describes the functional model for WebAuthn conformant authenticators, including their signature and attestation functionality.

0037 Status of this document

0038

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  
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 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 CredentialCreationOptions Extension
- 0106 2. 4.1.2 CredentialRequestOptions Extension
- 0107 3. 4.1.3 Create a new credential - PublicKeyCredential's
- 0108

- 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

0065 This section describes the status of this document at the time of its  
0066 publication. Other documents may supersede this document. A list of  
0067 current W3C publications and the latest revision of this technical  
0068 report can be found in the W3C technical reports index at  
0069 <http://www.w3.org/TR/>.

0070  
0071 This document was published by the Web Authentication Working Group as  
0072 an **Editors' Draft**. This document is intended to become a W3C  
0073 Recommendation. Feedback and comments on this specification are  
0074 welcome. Please use Github issues. Discussions may also be found in the  
0075 public-webauthn@w3.org archives.  
0076

0077 Publication as an **Editors' Draft** does not imply endorsement by the W3C  
0078 Membership. This is a draft document and may be updated, replaced or  
0079 obsoleted by other documents at any time. It is inappropriate to cite  
0080 this document as other than work in progress.

0081  
0082 This document was produced by a group operating under the 5 February  
0083 2004 W3C Patent Policy. W3C maintains a public list of any patent  
0084 disclosures made in connection with the deliverables of the group; that  
0085 page also includes instructions for disclosing a patent. An individual  
0086 who has actual knowledge of a patent which the individual believes  
0087 contains Essential Claim(s) must disclose the information in accordance  
0088 with section 6 of the W3C Patent Policy.

0089  
0090 This document is governed by the 1 March 2017 W3C Process Document.  
0091

0092 Table of Contents

- 0093 1. 1 Introduction
- 0094 1. 1.1 Use Cases
- 0095 1. 1.1.1 Registration
- 0096 2. 1.1.2 Authentication
- 0097 3. 1.1.3 Other use cases and configurations
- 0098 2. 2 Conformance
- 0099 1. 2.1 User Agents
- 0100 2. 2.2 Authenticators
- 0101 3. 2.3 Relying Parties
- 0102 3. 3 Dependencies
- 0103 4. 4 Terminology
- 0104 5. 5 Web Authentication API
- 0105 1. 5.1 PublicKeyCredential Interface
- 0106 1. 5.1.1 CredentialCreationOptions Extension
- 0107 2. 5.1.2 CredentialRequestOptions Extension
- 0108 3. 5.1.3 Create a new credential - PublicKeyCredential's
- 0109 [[Create]](options) method
- 0110 4. 5.1.4 Use an existing credential to make an assertion
- 0111 1. 5.1.4.1 PublicKeyCredential's
- 0112 [[DiscoverFromExternalSource]](options) method
- 0113 5. 5.1.5 Store an existing credential -
- 0114 PublicKeyCredential's [[Store]](credential) method
- 0115 6. 5.1.6 Platform Authenticator Availability -
- 0116 PublicKeyCredential's isPlatformAuthenticatorAvailable()
- 0117 method
- 0118 2. 5.2 Authenticator Responses (interface AuthenticatorResponse)
- 0119 1. 5.2.1 Information about Public Key Credential (interface
- 0120 AuthenticatorAttestationResponse)
- 0121 2. 5.2.2 Web Authentication Assertion (interface
- 0122 AuthenticatorAssertionResponse)
- 0123 3. 5.3 Parameters for Credential Generation (dictionary
- 0124 PublicKeyCredentialParameters)
- 0125 4. 5.4 Options for Credential Creation (dictionary
- 0126 MakePublicKeyCredentialOptions)
- 0127 1. 5.4.1 Public Key Entity Description (dictionary
- 0128 PublicKeyCredentialEntity)
- 0129 2. 5.4.2 RP Parameters for Credential Generation (dictionary
- 0130 PublicKeyCredentialRpEntity)
- 0131 3. 5.4.3 User Account Parameters for Credential Generation
- 0132 (dictionary PublicKeyCredentialUserEntity)
- 0133 4. 5.4.4 Authenticator Selection Criteria (dictionary
- 0134

- 0130 AuthenticatorSelectionCriteria)
- 0131 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)
- 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 10.11 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

- 0135 AuthenticatorSelectionCriteria)
- 0136 5.5.4.5 Authenticator Attachment enumeration (enum)
- 0137 AuthenticatorAttachment)
- 0138 5.5.5 Options for Assertion Generation (dictionary)
- 0139 PublicKeyCredentialRequestOptions)
- 0140 6.5.6 Authentication Extensions (typedef)
- 0141 AuthenticationExtensions)
- 0142 7.5.7 Supporting Data Structures
- 0143 1.5.7.1 Client data used in WebAuthn signatures (dictionary)
- 0144 CollectedClientData)
- 0145 2.5.7.2 Credential Type enumeration (enum)
- 0146 PublicKeyCredentialType)
- 0147 3.5.7.3 Credential Descriptor (dictionary)
- 0148 PublicKeyCredentialDescriptor)
- 0149 4.5.7.4 Authenticator Transport enumeration (enum)
- 0150 AuthenticatorTransport)
- 0151 5.5.7.5 Cryptographic Algorithm Identifier (typedef)
- 0152 COSEAlgorithmIdentifier)
- 0153 6.6 WebAuthn Authenticator model
- 0154 1.6.1 Authenticator data
- 0155 1.6.1.1 Signature Counter Considerations
- 0156 2.6.2 Authenticator operations
- 0157 1.6.2.1 The authenticatorMakeCredential operation
- 0158 2.6.2.2 The authenticatorGetAssertion operation
- 0159 3.6.2.3 The authenticatorCancel operation
- 0160 3.6.3 Attestation
- 0161 1.6.3.1 Attestation data
- 0162 2.6.3.2 Attestation Statement Formats
- 0163 3.6.3.3 Attestation Types
- 0164 4.6.3.4 Generating an Attestation Object
- 0165 5.6.3.5 Security Considerations
- 0166 1.6.3.5.1 Privacy
- 0167 2.6.3.5.2 Attestation Certificate and Attestation
- 0168 Certificate CA Compromise
- 0169 3.6.3.5.3 Attestation Certificate Hierarchy
- 0170 7.7 Relying Party Operations
- 0171 1.7.1 Registering a new credential
- 0172 2.7.2 Verifying an authentication assertion
- 0173 8.8 Defined Attestation Statement Formats
- 0174 1.8.1 Attestation Statement Format Identifiers
- 0175 2.8.2 Packed Attestation Statement Format
- 0176 1.8.2.1 Packed attestation statement certificate
- 0177 requirements
- 0178 3.8.3 TPM Attestation Statement Format
- 0179 1.8.3.1 TPM attestation statement certificate requirements
- 0180 4.8.4 Android Key Attestation Statement Format
- 0181 5.8.5 Android SafetyNet Attestation Statement Format
- 0182 6.8.6 FIDO U2F Attestation Statement Format
- 0183 9.9 WebAuthn Extensions
- 0184 1.9.1 Extension Identifiers
- 0185 2.9.2 Defining extensions
- 0186 3.9.3 Extending request parameters
- 0187 4.9.4 Client extension processing
- 0188 5.9.5 Authenticator extension processing
- 0189 6.9.6 Example Extension
- 0190 10.10 Defined Extensions
- 0191 1.10.1 FIDO AppId Extension (appid)
- 0192 2.10.2 Simple Transaction Authorization Extension (txAuthSimple)
- 0193 3.10.3 Generic Transaction Authorization Extension
- 0194 (txAuthGeneric)
- 0195 4.10.4 Authenticator Selection Extension (authnSel)
- 0196 5.10.5 Supported Extensions Extension (exts)
- 0197 6.10.6 User Verification Index Extension (uvi)
- 0198 7.10.7 Location Extension (loc)
- 0199 8.10.8 User Verification Method Extension (uvm)
- 0200 11.11 IANA Considerations
- 0201 1.11.1 WebAuthn Attestation Statement Format Identifier
- 0202 Registrations
- 0203 2.11.2 WebAuthn Extension Identifier Registrations
- 0204 3.11.3 COSE Algorithm Registrations



0195 **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**  
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

0205 **12. 12 Sample scenarios**  
0206 1. **12.1 Registration**  
0207 2. **12.2 Registration Specifically with Platform Authenticator**  
0208 3. **12.3 Authentication**  
0209 4. **12.4 Decommissioning**  
0210 **13. 13 Acknowledgements**  
0211 **14. Index**  
0212 1. Terms defined by this specification  
0213 2. Terms defined by reference  
0214 **15. References**  
0215 1. Normative References  
0216 2. Informative References  
0217 **16. IDL Index**

0218  
0219 **1. Introduction**  
0220  
0221 This section is not normative.  
0222  
0223 This specification defines an API enabling the creation and use of  
0224 strong, attested, scoped, public key-based credentials by web  
0225 applications, for the purpose of strongly authenticating users. A  
0226 public key credential is created and stored by an authenticator at the  
0227 behest of a Relying Party, subject to user consent. Subsequently, the  
0228 public key credential can only be accessed by origins belonging to that  
0229 Relying Party. This scoping is enforced jointly by conforming User  
0230 Agents and authenticators. Additionally, privacy across Relying Parties  
0231 is maintained; Relying Parties are not able to detect any properties,  
0232 or even the existence, of credentials scoped to other Relying Parties.  
0233  
0234 Relying Parties employ the Web Authentication API during two distinct,  
0235 but related, ceremonies involving a user. The first is Registration,  
0236 where a public key credential is created on an authenticator, and  
0237 associated by a Relying Party with the present user's account (the  
0238 account may already exist or may be created at this time). The second  
0239 is Authentication, where the Relying Party is presented with an  
0240 Authentication Assertion proving the presence and consent of the user  
0241 who registered the public key credential. Functionally, the Web  
0242 Authentication API comprises a PublicKeyCredential which extends the  
0243 Credential Management API [CREDENTIAL-MANAGEMENT-1], and infrastructure  
0244 which allows those credentials to be used with  
0245 navigator.credentials.create() and navigator.credentials.get(). The  
0246 former is used during Registration, and the latter during  
0247 Authentication.  
0248  
0249 Broadly, compliant authenticators protect public key credentials, and  
0250 interact with user agents to implement the Web Authentication API. Some  
0251 authenticators may run on the same computing device (e.g., smart phone,  
0252 tablet, desktop PC) as the user agent is running on. For instance, such  
0253 an authenticator might consist of a Trusted Execution Environment (TEE)  
0254 applet, a Trusted Platform Module (TPM), or a Secure Element (SE)  
0255 integrated into the computing device in conjunction with some means for  
0256 user verification, along with appropriate platform software to mediate  
0257 access to these components' functionality. Other authenticators may  
0258 operate autonomously from the computing device running the user agent,  
0259 and be accessed over a transport such as Universal Serial Bus (USB),  
0260 Bluetooth Low Energy (BLE) or Near Field Communications (NFC).  
0261  
0262 **1.1. Use Cases**  
0263  
0264 The below use case scenarios illustrate use of two very different types  
0265 of authenticators, as well as outline further scenarios. Additional  
0266 scenarios, including sample code, are given later in **12 Sample**  
0267 **scenarios**.  
0268  
0269 **1.1.1. Registration**  
0270  
0271 \* On a phone:  
0272 + User navigates to example.com in a browser and signs in to an  
0273 existing account using whatever method they have been using  
0274 (possibly a legacy method such as a password), or creates a

026E new account.  
 027C + 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.  
 027E + Website shows message, "Registration complete."

### 1.1.2. Authentication

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

### 1.1.3. Other use cases and configurations

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

- \* A user navigates to example.com on their laptop, is guided through a flow to create and register a credential on their phone.
- \* A user obtains an discrete, roaming authenticator, such as a "fob" with USB or USB+NFC/BLE connectivity options, loads example.com in their browser on a laptop or phone, and is guided 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

027E new account.  
 027E + The phone prompts, "Do you want to register this device with  
 0277 example.com?"  
 027E + User agrees.  
 027E + The phone prompts the user for a previously configured  
 0280 authorization gesture (PIN, biometric, etc.); the user  
 0281 provides this.  
 0282 + Website shows message, "Registration complete."

### 1.1.2. Authentication

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

### 1.1.3. Other use cases and configurations

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

- \* A user navigates to example.com on their laptop, is guided through a flow to create and register a credential on their phone.
- \* A user obtains an discrete, roaming authenticator, such as a "fob" with USB or USB+NFC/BLE connectivity options, loads example.com in their browser on a laptop or phone, and is guided 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 **three conformance classes. Each of these classes is specified so that conforming members of the class are secure against non-conforming or hostile members of the other classes.**

### 2.1. User Agents

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

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

0330  
0331  
0332  
0333  
0334  
0335  
0336  
0337  
0338  
0339  
0340  
0341  
0342  
0343  
0344  
0345  
0346  
0347  
0348  
0349  
0350  
0351  
0352  
0353  
0354  
0355  
0356  
0357  
0358  
0359  
0360  
0361  
0362  
0363  
0364  
0365  
0366  
0367  
0368  
0369  
0370  
0371  
0372  
0373  
0374  
0375  
0376  
0377  
0378  
0379  
0380  
0381  
0382

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

**Base64url encoding**

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

**CBOR**

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

**CDDL**

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

**COSE**

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

**Credential Management**

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

**DOM**

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

**ECMAScript**

%ArrayBuffer% is defined in [ECMAScript].

**HTML**

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

**Web IDL**

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

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

**3. Terminology**

0335  
0336  
0337  
0338  
0339  
0340  
0341  
0342  
0343  
0344  
0345  
0346  
0347  
0348  
0349  
0350  
0351  
0352  
0353  
0354  
0355  
0356  
0357  
0358  
0359  
0360  
0361  
0362  
0363  
0364  
0365  
0366  
0367  
0368  
0369  
0370  
0371  
0372  
0373  
0374  
0375  
0376  
0377  
0378  
0379  
0380  
0381  
0382  
0393  
0394  
0395  
0396  
0397  
0398  
0399  
0400  
0401  
0402  
0403  
0404

**2.2. Authenticators**

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

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

**2.3. Relying Parties**

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

**3. Dependencies**

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

**Base64url encoding**

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

**CBOR**

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

**CDDL**

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

**COSE**

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

**Credential Management**

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

**DOM**

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

**ECMAScript**

%ArrayBuffer% is defined in [ECMAScript].

**HTML**

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

**Web IDL**

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

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

**4. 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  
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

0405  
0406 **Assertion**  
0407 See Authentication Assertion.  
0408

0409 **Attestation**  
0410 Generally, attestation is a statement serving to bear witness,  
0411 confirm, or authenticate. In the WebAuthn context, attestation  
0412 is employed to attest to the provenance of an authenticator and  
0413 the data it emits; including, for example: credential IDs,  
0414 credential key pairs, signature counters, etc. An attestation  
0415 statement is conveyed in an attestation object during  
0416 registration. See also 6.3 Attestation and Figure 3.  
0417

0418 **Attestation Certificate**  
0419 A X.509 Certificate for the attestation key pair used by an  
0420 authenticator to attest to its manufacture and capabilities. At  
0421 registration time, the authenticator uses the attestation  
0422 private key to sign the Relying Party-specific credential public  
0423 key (and additional data) that it generates and returns via the  
0424 authenticatorMakeCredential operation. Relying Parties use the  
0425 attestation public key conveyed in the attestation certificate  
0426 to verify the attestation signature. Note that in the case of  
0427 self attestation, the authenticator has no distinct attestation  
0428 key pair nor attestation certificate, see self attestation for  
0429 details.  
0430

0431 **Authentication**  
0432 The ceremony where a user, and the user's computing device(s)  
0433 (containing at least one authenticator) work in concert to  
0434 cryptographically prove to an Relying Party that the user  
0435 controls the credential private key associated with a  
0436 previously-registered public key credential (see Registration).  
0437 Note that this typically includes employing a test of user  
0438 presence or user verification.  
0439

0440 **Authentication Assertion**  
0441 The cryptographically signed AuthenticatorAssertionResponse  
0442 object returned by an authenticator as the result of a  
0443 authenticatorGetAssertion operation.  
0444

0445 **Authenticator**  
0446 A cryptographic **entity** used by a WebAuthn Client to (i) generate  
0447 a public key credential and register it with a Relying Party,  
0448 and (ii) **authenticate by potentially verifying the user, and**  
0449 **then cryptographically signing and returning, in the form of an**  
0450 **Authentication Assertion, a challenge and other data presented**  
0451 **by a Relying Party (in concert with the WebAuthn Client).**  
0452

0453 **Authorization Gesture**  
0454 An authorization gesture is a physical interaction performed by  
0455 a user with an authenticator as part of a ceremony, such as  
0456 registration or authentication. By making such an authorization  
0457 gesture, a user provides consent for (i.e., authorizes) a  
0458 ceremony to proceed. This may involve user verification if the  
0459 employed authenticator is capable, or it may involve a simple  
0460 test of user presence.  
0461

0462 **Biometric Recognition**  
0463 The automated recognition of individuals based on their  
0464 biological and behavioral characteristics  
0465 [ISOBiometricVocabulary].  
0466

0467 **Ceremony**  
0468 The concept of a ceremony [Ceremony] is an extension of the  
0469 concept of a network protocol, with human nodes alongside  
0470 computer nodes and with communication links that include user  
0471 interface(s), human-to-human communication, and transfers of  
0472 physical objects that carry data. What is out-of-band to a  
0473 protocol is in-band to a ceremony. In this specification,  
0474 Registration and Authentication are ceremonies, and an

0455 authorization gesture is often a component of those ceremonies.  
0456  
0457 **Client**  
0458 See Conforming User Agent.  
0459  
0460 **Client-Side**  
0461 This refers in general to the combination of the user's platform  
0462 device, user agent, authenticators, and everything gluing it all  
0463 together.  
0464  
0465 **Client-side-resident Credential Private Key**  
0466 A Client-side-resident Credential Private Key is stored either  
0467 on the client platform, or in some cases on the authenticator  
0468 itself, e.g., in the case of a discrete first-factor roaming  
0469 authenticator. Such client-side credential private key storage  
0470 has the property that the authenticator is able to select the  
0471 credential private key given only an RP ID, possibly with user  
0472 assistance (e.g., by providing the user a pick list of  
0473 credentials associated with the RP ID). By definition, the  
0474 private key is always exclusively controlled by the  
0475 Authenticator. In the case of a Client-side-resident Credential  
0476 Private Key, the Authenticator might offload storage of wrapped  
0477 key material to the client platform, but the client platform is  
0478 not expected to offload the key storage to remote entities (e.g.  
0479 RP Server).  
0480  
0481 **Conforming User Agent**  
0482 A user agent implementing, in conjunction with the underlying  
0483 platform, the Web Authentication API and algorithms given in  
0484 this specification, and handling communication between  
0485 authenticators and Relying Parties.  
0486  
0487 **Credential Public Key**  
0488 The public key portion of an Relying Party-specific credential  
0489 key pair, generated by an authenticator and returned to an  
0490 Relying Party at registration time (see also public key  
0491 credential). The private key portion of the credential key pair  
0492 is known as the credential private key. Note that in the case of  
0493 self attestation, the credential key pair is also used as the  
0494 attestation key pair, see self attestation for details.  
0495  
0496 **Rate Limiting**  
0497 The process (also known as throttling) by which an authenticator  
0498 implements controls against brute force attacks by limiting the  
0499 number of consecutive failed authentication attempts within a  
0500 given period of time. If the limit is reached, the authenticator  
0501 should impose a delay that increases exponentially with each  
0502 successive attempt, or disable the current authentication  
0503 modality and offer a different authentication factor if  
0504 available. Rate limiting is often implemented as an aspect of  
0505 user verification.  
0506  
0507 **Registration**  
0508 The ceremony where a user, a Relying Party, and the user's  
0509 computing device(s) (containing at least one authenticator) work  
0510 in concert to create a public key credential and associate it  
0511 with the user's Relying Party account. Note that this typically  
0512 includes employing a test of user presence or user verification.  
0513  
0514 **Relying Party**  
0515 The entity whose web application utilizes the Web Authentication  
0516 API to register and authenticate users. See Registration and  
0517 Authentication, respectively.  
0518  
0519 **Note:** While the term Relying Party is used in other contexts  
0520 (e.g., X.509 and OAuth), an entity acting as a Relying Party in  
0521 one context is not necessarily a Relying Party in other  
0522 contexts.  
0523  
0524 **Relying Party Identifier**

0475 authorization gesture is often a component of those ceremonies.  
0476  
0477 **Client**  
0478 See Conforming User Agent.  
0479  
0480 **Client-Side**  
0481 This refers in general to the combination of the user's platform  
0482 device, user agent, authenticators, and everything gluing it all  
0483 together.  
0484  
0485 **Client-side-resident Credential Private Key**  
0486 A Client-side-resident Credential Private Key is stored either  
0487 on the client platform, or in some cases on the authenticator  
0488 itself, e.g., in the case of a discrete first-factor roaming  
0489 authenticator. Such client-side credential private key storage  
0490 has the property that the authenticator is able to select the  
0491 credential private key given only an RP ID, possibly with user  
0492 assistance (e.g., by providing the user a pick list of  
0493 credentials associated with the RP ID). By definition, the  
0494 private key is always exclusively controlled by the  
0495 Authenticator. In the case of a Client-side-resident Credential  
0496 Private Key, the Authenticator might offload storage of wrapped  
0497 key material to the client platform, but the client platform is  
0498 not expected to offload the key storage to remote entities (e.g.  
0499 RP Server).  
0500  
0501 **Conforming User Agent**  
0502 A user agent implementing, in conjunction with the underlying  
0503 platform, the Web Authentication API and algorithms given in  
0504 this specification, and handling communication between  
0505 authenticators and Relying Parties.  
0506  
0507 **Credential Public Key**  
0508 The public key portion of an Relying Party-specific credential  
0509 key pair, generated by an authenticator and returned to an  
0510 Relying Party at registration time (see also public key  
0511 credential). The private key portion of the credential key pair  
0512 is known as the credential private key. Note that in the case of  
0513 self attestation, the credential key pair is also used as the  
0514 attestation key pair, see self attestation for details.  
0515  
0516 **Rate Limiting**  
0517 The process (also known as throttling) by which an authenticator  
0518 implements controls against brute force attacks by limiting the  
0519 number of consecutive failed authentication attempts within a  
0520 given period of time. If the limit is reached, the authenticator  
0521 should impose a delay that increases exponentially with each  
0522 successive attempt, or disable the current authentication  
0523 modality and offer a different authentication factor if  
0524 available. Rate limiting is often implemented as an aspect of  
0525 user verification.  
0526  
0527 **Registration**  
0528 The ceremony where a user, a Relying Party, and the user's  
0529 computing device(s) (containing at least one authenticator) work  
0530 in concert to create a public key credential and associate it  
0531 with the user's Relying Party account. Note that this typically  
0532 includes employing a test of user presence or user verification.  
0533  
0534 **Relying Party**  
0535 The entity whose web application utilizes the Web Authentication  
0536 API to register and authenticate users. See Registration and  
0537 Authentication, respectively.  
0538  
0539 **Note:** While the term Relying Party is used in other contexts  
0540 (e.g., X.509 and OAuth), an entity acting as a Relying Party in  
0541 one context is not necessarily a Relying Party in other  
0542 contexts.  
0543  
0544 **Relying Party Identifier**



0523 **RP ID**  
 0524 A valid domain string that identifies the Relying Party on whose  
 0525 behalf a given registration or authentication ceremony is being  
 0526 performed. A public key credential can only be used for  
 0527 authentication with the same entity (as identified by RP ID) it  
 0528 was registered with. By default, the RP ID for a WebAuthn  
 0529 operation is set to the caller's origin's effective domain. This  
 0530 default MAY be overridden by the caller, as long as the  
 0531 caller-specified RP ID value is a registrable domain suffix of  
 0532 or is equal to the caller's origin's effective domain. See also  
 0533 4.1.3 Create a new credential - PublicKeyCredential's  
 0534 [[Create]](options) method and 4.1.4 Use an existing credential  
 0535 to make an assertion - PublicKeyCredential's  
 0536 [[DiscoverFromExternalSource]](options) method.  
 0537  
 0538 Note: A Public key credential's scope is for a Relying Party's  
 0539 origin, with the following restrictions and relaxations:  
 0540  
 0541 + The scheme is always https (i.e., a restriction), and,  
 0542 + the host may be equal to the Relying Party's origin's  
 0543 effective domain, or it may be equal to a registrable domain  
 0544 suffix of the Relying Party's origin's effective domain (i.e.,  
 0545 an available relaxation), and,  
 0546 + all (TCP) ports on that host (i.e., a relaxation).  
 0547  
 0548 This is done in order to match the behavior of pervasively  
 0549 deployed ambient credentials (e.g., cookies, [RFC6265]). Please  
 0550 note that this is a greater relaxation of "same-origin"  
 0551 restrictions than what document.domain's setter provides.  
 0552  
 0553 **Public Key Credential**  
 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 A test of user presence is a simple form of authorization  
 0572 gesture and technical process where a user interacts with an  
 0573 authenticator by (typically) simply touching it (other  
 0574 modalities may also exist), yielding a boolean result. Note that  
 0575 this does not constitute user verification because a user  
 0576 presence test, by definition, is not capable of biometric  
 0577 recognition, nor does it involve the presentation of a shared  
 0578 secret such as a password or PIN.  
 0579  
 0580 **User Consent**  
 0581 User consent means the user agrees with what they are being  
 0582 asked, i.e., it encompasses reading and understanding prompts.  
 0583 An authorization gesture is a ceremony component often employed  
 0584 to indicate user consent.  
 0585

0546 **RP ID**  
 0547 A valid domain string that identifies the Relying Party on whose  
 0548 behalf a given registration or authentication ceremony is being  
 0549 performed. A public key credential can only be used for  
 0550 authentication with the same entity (as identified by RP ID) it  
 0551 was registered with. By default, the RP ID for a WebAuthn  
 0552 operation is set to the caller's origin's effective domain. This  
 0553 default MAY be overridden by the caller, as long as the  
 0554 caller-specified RP ID value is a registrable domain suffix of  
 0555 or is equal to the caller's origin's effective domain. See also  
 0556 5.1.3 Create a new credential - PublicKeyCredential's  
 0557 [[Create]](options) method and 5.1.4 Use an existing credential  
 0558 to make an assertion.  
 0559  
 0560 Note: A Public key credential's scope is for a Relying Party's  
 0561 origin, with the following restrictions and relaxations:  
 0562  
 0563 + The scheme is always https (i.e., a restriction), and,  
 0564 + the host may be equal to the Relying Party's origin's  
 0565 effective domain, or it may be equal to a registrable domain  
 0566 suffix of the Relying Party's origin's effective domain (i.e.,  
 0567 an available relaxation), and,  
 0568 + all (TCP) ports on that host (i.e., a relaxation).  
 0569  
 0570 This is done in order to match the behavior of pervasively  
 0571 deployed ambient credentials (e.g., cookies, [RFC6265]). Please  
 0572 note that this is a greater relaxation of "same-origin"  
 0573 restrictions than what document.domain's setter provides.  
 0574  
 0575 **Public Key Credential**  
 0576 Generically, a credential is data one entity presents to another  
 0577 in order to authenticate the former to the latter [RFC4949]. A  
 0578 WebAuthn public key credential is a { identifier, type } pair  
 0579 identifying authentication information established by the  
 0580 authenticator and the Relying Party, together, at registration  
 0581 time. The authentication information consists of an asymmetric  
 0582 key pair, where the public key portion is returned to the  
 0583 Relying Party, who then stores it in conjunction with the  
 0584 present user's account. The authenticator maps the private key  
 0585 portion to the Relying Party's RP ID and stores it.  
 0586 Subsequently, only that Relying Party, as identified by its RP  
 0587 ID, is able to employ the public key credential in  
 0588 authentication ceremonies, via the get() method. The Relying  
 0589 Party uses its stored copy of the credential public key to  
 0590 verify the resultant authentication assertion.  
 0591  
 0592 **Test of User Presence**  
 0593 A test of user presence is a simple form of authorization  
 0594 gesture and technical process where a user interacts with an  
 0595 authenticator by (typically) simply touching it (other  
 0596 modalities may also exist), yielding a boolean result. Note that  
 0597 this does not constitute user verification because a user  
 0598 presence test, by definition, is not capable of biometric  
 0599 recognition, nor does it involve the presentation of a shared  
 0600 secret such as a password or PIN.  
 0601  
 0602 **User Consent**  
 0603 User consent means the user agrees with what they are being  
 0604 asked, i.e., it encompasses reading and understanding prompts.  
 0605 An authorization gesture is a ceremony component often employed  
 0606 to indicate user consent.  
 0607  
 0608 **User Handle**  
 0609 The user handle is specified by a Relying Party and is a unique  
 0610 identifier for a user account with that Relying Party. A user  
 0611 handle is an opaque byte sequence with a maximum size of 64  
 0612 bytes.  
 0613 The user handle is not meant to be displayed to the user, but is

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

0600 **User Present**  
0601 UP  
0602 Upon successful completion of a user presence test, the user is  
0603 said to be "present".

0604 **User Verified**  
0605 UV  
0606 Upon successful completion of a user verification process, the  
0607 user is said to be "verified".

0608 **WebAuthn Client**  
0609 Also referred to herein as simply a client. See also Conforming  
0610 User Agent.

0611 **4. Web Authentication API**  
0612 This section normatively specifies the API for creating and using  
0613 public key credentials. The basic idea is that the credentials belong  
0614 to the user and are managed by an authenticator, with which the Relying  
0615 Party interacts through the client (consisting of the browser and  
0616 underlying OS platform). Scripts can (with the user's consent) request  
0617 the browser to create a new credential for future use by the Relying  
0618 Party. Scripts can also request the user's permission to perform  
0619 authentication operations with an existing credential. All such  
0620 operations are performed in the authenticator and are mediated by the  
0621 browser and/or platform on the user's behalf. At no point does the  
0622 script get access to the credentials themselves; it only gets  
0623 information about the credentials in the form of objects.

0624 In addition to the above script interface, the authenticator may  
0625 implement (or come with client software that implements) a user  
0626 interface for management. Such an interface may be used, for example,  
0627 to reset the authenticator to a clean state or to inspect the current  
0628 state of the authenticator. In other words, such an interface is  
0629 similar to the user interfaces provided by browsers for managing user  
0630 state such as history, saved passwords and cookies. Authenticator  
0631 management actions such as credential deletion are considered to be the  
0632 responsibility of such a user interface and are deliberately omitted  
0633 from the API exposed to scripts.

0634 The security properties of this API are provided by the client and the  
0635 authenticator working together. The authenticator, which holds and  
0636 manages credentials, ensures that all operations are scoped to a  
0637 particular origin, and cannot be replayed against a different origin,  
0638 by incorporating the origin in its responses. Specifically, as defined  
0639 in 5.2 Authenticator operations, the full origin of the requester is  
0640 included, and signed over, in the attestation object produced when a  
0641 new credential is created as well as in all assertions produced by  
0642 WebAuthn credentials.

0643 Additionally, to maintain user privacy and prevent malicious Relying

0614 used by the Relying Party to control the number of credentials -  
0615 an authenticator will never contain more than one credential for  
0616 a given Relying Party under the same user handle.

0617 **User Verification**  
0618 The technical process by which an authenticator locally  
0619 authorizes the invocation of the authenticatorMakeCredential and  
0620 authenticatorGetAssertion operations. User verification may be  
0621 instigated through various authorization gesture modalities; for  
0622 example, through a touch plus pin code, password entry, or  
0623 biometric recognition (e.g., presenting a fingerprint)  
0624 [ISOBiometricVocabulary]. The intent is to be able to  
0625 distinguish individual users. Note that invocation of the  
0626 authenticatorMakeCredential and authenticatorGetAssertion  
0627 operations implies use of key material managed by the  
0628 authenticator. Note that for security, user verification and use  
0629 of credential private keys must occur within a single logical  
0630 security boundary defining the authenticator.

0631 **User Present**  
0632 UP  
0633 Upon successful completion of a user presence test, the user is  
0634 said to be "present".

0635 **User Verified**  
0636 UV  
0637 Upon successful completion of a user verification process, the  
0638 user is said to be "verified".

0639 **WebAuthn Client**  
0640 Also referred to herein as simply a client. See also Conforming  
0641 User Agent.

0642 **5. Web Authentication API**  
0643 This section normatively specifies the API for creating and using  
0644 public key credentials. The basic idea is that the credentials belong  
0645 to the user and are managed by an authenticator, with which the Relying  
0646 Party interacts through the client (consisting of the browser and  
0647 underlying OS platform). Scripts can (with the user's consent) request  
0648 the browser to create a new credential for future use by the Relying  
0649 Party. Scripts can also request the user's permission to perform  
0650 authentication operations with an existing credential. All such  
0651 operations are performed in the authenticator and are mediated by the  
0652 browser and/or platform on the user's behalf. At no point does the  
0653 script get access to the credentials themselves; it only gets  
0654 information about the credentials in the form of objects.

0655 In addition to the above script interface, the authenticator may  
0656 implement (or come with client software that implements) a user  
0657 interface for management. Such an interface may be used, for example,  
0658 to reset the authenticator to a clean state or to inspect the current  
0659 state of the authenticator. In other words, such an interface is  
0660 similar to the user interfaces provided by browsers for managing user  
0661 state such as history, saved passwords and cookies. Authenticator  
0662 management actions such as credential deletion are considered to be the  
0663 responsibility of such a user interface and are deliberately omitted  
0664 from the API exposed to scripts.

0665 The security properties of this API are provided by the client and the  
0666 authenticator working together. The authenticator, which holds and  
0667 manages credentials, ensures that all operations are scoped to a  
0668 particular origin, and cannot be replayed against a different origin,  
0669 by incorporating the origin in its responses. Specifically, as defined  
0670 in 6.2 Authenticator operations, the full origin of the requester is  
0671 included, and signed over, in the attestation object produced when a  
0672 new credential is created as well as in all assertions produced by  
0673 WebAuthn credentials.

0674 Additionally, to maintain user privacy and prevent malicious Relying

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

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

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

#### 4.1. PublicKeyCredential Interface

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

```
[SecureContext]
interface PublicKeyCredential : Credential {
  [SameObject] readonly attribute ArrayBuffer rawId;
  [SameObject] readonly attribute AuthenticatorResponse response;
  [SameObject] readonly attribute AuthenticationExtensions clientExtensionResults;
};
```

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

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

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

**clientExtensionResults**, of type `AuthenticationExtensions`, **readonly**  
This attribute contains a map containing extension identifier -> client extension output entries produced by the extension's client extension processing.

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

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

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

**[[identifier]]**  
This internal slot contains an identifier for the credential, chosen by the platform with help from the authenticator. This

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

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

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

#### 5.1. PublicKeyCredential Interface

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

```
[SecureContext]
interface PublicKeyCredential : Credential {
  [SameObject] readonly attribute ArrayBuffer rawId;
  [SameObject] readonly attribute AuthenticatorResponse response;
  [SameObject] readonly attribute AuthenticationExtensions clientExtensionResults;
};
```

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

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

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

**clientExtensionResults**, of type `AuthenticationExtensions`, **readonly**  
This attribute contains a map containing extension identifier -> client extension output entries produced by the extension's client extension processing.

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

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

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

**[[identifier]]**  
This internal slot contains an identifier for the credential, chosen by the platform with help from the authenticator. This



0722 identifier is used to look up credentials for use, and is  
 0723 therefore expected to be globally unique with high probability  
 0724 across all credentials of the same type, across all  
 0725 authenticators. This API does not constrain the format or length  
 0726 of this identifier, except that it must be sufficient for the  
 0727 platform to uniquely select a key. For example, an authenticator  
 0728 without on-board storage may create identifiers containing a  
 0729 credential private key wrapped with a symmetric key that is  
 0730 burned into the authenticator.  
 0731  
 0732 PublicKeyCredential's interface object inherits Credential's  
 0733 implementation of `[[CollectFromCredentialStore]](options)` and  
 0734 `[[Store]](credential)`, and defines its own implementation of  
 0735 `[[DiscoverFromExternalSource]](options)` and `[[Create]](options)`.  
 0736  
 0737 **4.1.1. CredentialCreationOptions Extension**  
 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.

0754 identifier is used to look up credentials for use, and is  
 0755 therefore expected to be globally unique with high probability  
 0756 across all credentials of the same type, across all  
 0757 authenticators. This API does not constrain the format or length  
 0758 of this identifier, except that it must be sufficient for the  
 0759 platform to uniquely select a key. For example, an authenticator  
 0760 without on-board storage may create identifiers containing a  
 0761 credential private key wrapped with a symmetric key that is  
 0762 burned into the authenticator.  
 0763  
 0764 PublicKeyCredential's interface object inherits Credential's  
 0765 implementation of `[[CollectFromCredentialStore]](options)` and  
 0766 `[[Store]](credential)`, and defines its own implementation of  
 0767 `[[DiscoverFromExternalSource]](options)` and `[[Create]](options)`.  
 0768  
 0769 **5.1.1. CredentialCreationOptions Extension**  
 0770  
 0771 To support registration via `navigator.credentials.create()`, this  
 0772 document extends the `CredentialCreationOptions` dictionary as follows:  
 0773 partial dictionary `CredentialCreationOptions` {  
 0774   `MakePublicKeyCredentialOptions`   `publicKey`;  
 0775 };  
 0776  
 0777 **5.1.2. CredentialRequestOptions Extension**  
 0778  
 0779 To support obtaining assertions via `navigator.credentials.get()`, this  
 0780 document extends the `CredentialRequestOptions` dictionary as follows:  
 0781 partial dictionary `CredentialRequestOptions` {  
 0782   `PublicKeyCredentialRequestOptions`   `publicKey`;  
 0783 };  
 0784  
 0785 **5.1.3. Create a new credential - PublicKeyCredential's `[[Create]](options)`  
 0786 method**  
 0787  
 0788 PublicKeyCredential's interface object's implementation of the  
 0789 `[[Create]](options)` method allows scripts to call  
 0790 `navigator.credentials.create()` to request the creation of a new  
 0791 credential key pair and `PublicKeyCredential`, managed by an  
 0792 authenticator. **On success, the returned promise will be resolved with a**  
 0793 `PublicKeyCredential` containing an `AuthenticatorAttestationResponse`  
 0794 object.  
 0795  
 0796 Note: This algorithm is synchronous; the Promise resolution/rejection  
 0797 is handled by `navigator.credentials.create()`.  
 0798  
 0799 This method accepts a single argument:  
 0800  
 0801 `options`  
 0802 This argument is a `CredentialCreationOptions` object whose  
 0803 `options.publicKey` member contains a  
 0804 `MakePublicKeyCredentialOptions` object specifying the desired  
 0805 attributes of the to-be-created public key credential.  
 0806  
 0807 When this method is invoked, the user agent MUST execute the following  
 0808 algorithm:  
 0809 1. Assert: `options.publicKey` is present.  
 0810 2. Let `options` be the value of `options.publicKey`.  
 0811 3. If any of the name member of `options.rp`, the name member of  
 0812 `options.user`, the `displayName` member of `options.user`, or the `id`  
 0813 member of `options.user` are not present, return a `TypeError` simple  
 0814 exception.  
 0815 4. If the `timeout` member of `options` is present, check if its value  
 0816 lies within a reasonable range as defined by the platform and if  
 0817 not, correct it to the closest value lying within that range. Set  
 0818 `adjustedTimeout` to this adjusted value. If the `timeout` member of  
 0819 `options` is not present, then set `adjustedTimeout` to a  
 0820 platform-specific default.  
 0821 5. Let `global` be the `PublicKeyCredential`'s interface object's  
 0822 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  
0797 7. Let effectiveDomain be the callerOrigin's effective domain. If  
0798 effective domain is not a valid domain, then return a DOMException  
0799 whose name is "SecurityError" and terminate this algorithm.  
0800 Note: An effective domain may resolve to a host, which can be  
0801 represented in various manners, such as domain, ipv4 address, ipv6  
0802 address, opaque host, or empty host. Only the domain format of host  
0803 is allowed here.  
0804  
0805 8. Let rpId be effectiveDomain.  
0806  
0807 9. If options rp.id is present:  
0808 1. If options rp.id is not a registrable domain suffix of and is  
0809 not equal to effectiveDomain, return a DOMException whose name  
0810 is "SecurityError", and terminate this algorithm.  
0811  
0812 2. Set rpId to options rp.id.  
0813 Note: rpId represents the caller's RP ID. The RP ID defaults  
0814 to being the caller's origin's effective domain unless the  
0815 caller has explicitly set options rp.id when calling create().  
0816  
0817 10. Let credTypesAndPubKeyAlgs be a new list whose items are pairs of  
0818 PublicKeyCredentialType and a COSEAlgorithmIdentifier.  
0819  
0820 11. For each current of options pubKeyCredParams:  
0821 1. If current.type does not contain a PublicKeyCredentialType  
0822 supported by this implementation, then continue.  
0823  
0824 2. Let alg be current.alg.  
0825  
0826 3. Append the pair of current.type and alg to  
0827 credTypesAndPubKeyAlgs.  
0828  
0829 12. If credTypesAndPubKeyAlgs is empty and options pubKeyCredParams is  
0830 not empty, cancel the timer started in step 2, return a  
0831 DOMException whose name is "NotSupportedError", and terminate this  
0832 algorithm.  
0833  
0834 13. Let clientExtensions be a new map and let authenticatorExtensions  
0835 be a new map.  
0836  
0837 14. If the extensions member of options is present, then for each  
0838 extensionId -> clientExtensionInput of options.extensions:  
0839 1. If extensionId is not supported by this client platform or is  
0840 not a registration extension, then continue.  
0841  
0842 2. Set clientExtensions[extensionId] to clientExtensionInput.  
0843  
0844 3. If extensionId is not an authenticator extension, then  
0845 continue.  
0846  
0847 4. Let authenticatorExtensionInput be the (CBOR) result of  
0848 running extensionId's client extension processing algorithm on  
0849 clientExtensionInput. If the algorithm returned an error,  
0850 continue.  
0851  
0852 5. Set authenticatorExtensions[extensionId] to the base64url  
0853 encoding of authenticatorExtensionInput.  
0854  
0855 15. Let collectedClientData be a new CollectedClientData instance whose  
0856 fields are:  
0857  
0858 challenge  
0859 The base64url encoding of options.challenge.  
0860  
0861 origin  
0862 The serialization of callerOrigin.  
0863  
0864 hashAlgorithm  
0865 The recognized algorithm name of the hash algorithm  
0866 selected by the client for generating the hash of the  
0867 serialized client data.  
0868  
0869 tokenBindingId  
0870 The Token Binding ID associated with callerOrigin, if one  
0871 is available.  
0872  
0873 clientExtensions  
0874 clientExtensions  
0875  
0876 authenticatorExtensions  
0877 authenticatorExtensions

0823 6. Let callerOrigin be the origin specified by this  
0824 PublicKeyCredential interface object's relevant settings object. If  
0825 callerOrigin is an opaque origin, return a DOMException whose name  
0826 is "NotAllowedError", and terminate this algorithm.  
0827  
0828 7. Let effectiveDomain be the callerOrigin's effective domain. If  
0829 effective domain is not a valid domain, then return a DOMException  
0830 whose name is "SecurityError" and terminate this algorithm.  
0831 Note: An effective domain may resolve to a host, which can be  
0832 represented in various manners, such as domain, ipv4 address, ipv6  
0833 address, opaque host, or empty host. Only the domain format of host  
0834 is allowed here.  
0835  
0836 8. Let rpId be effectiveDomain.  
0837  
0838 9. If options rp.id is present:  
0839 1. If options rp.id is not a registrable domain suffix of and is  
0840 not equal to effectiveDomain, return a DOMException whose name  
0841 is "SecurityError", and terminate this algorithm.  
0842  
0843 2. Set rpId to options rp.id.  
0844 Note: rpId represents the caller's RP ID. The RP ID defaults  
0845 to being the caller's origin's effective domain unless the  
0846 caller has explicitly set options rp.id when calling create().  
0847  
0848 10. Let credTypesAndPubKeyAlgs be a new list whose items are pairs of  
0849 PublicKeyCredentialType and a COSEAlgorithmIdentifier.  
0850  
0851 11. For each current of options pubKeyCredParams:  
0852 1. If current.type does not contain a PublicKeyCredentialType  
0853 supported by this implementation, then continue.  
0854  
0855 2. Let alg be current.alg.  
0856  
0857 3. Append the pair of current.type and alg to  
0858 credTypesAndPubKeyAlgs.  
0859  
0860 12. If credTypesAndPubKeyAlgs is empty and options pubKeyCredParams is  
0861 not empty, cancel the timer started in step 2, return a  
0862 DOMException whose name is "NotSupportedError", and terminate this  
0863 algorithm.  
0864  
0865 13. Let clientExtensions be a new map and let authenticatorExtensions  
0866 be a new map.  
0867  
0868 14. If the extensions member of options is present, then for each  
0869 extensionId -> clientExtensionInput of options.extensions:  
0870 1. If extensionId is not supported by this client platform or is  
0871 not a registration extension, then continue.  
0872  
0873 2. Set clientExtensions[extensionId] to clientExtensionInput.  
0874  
0875 3. If extensionId is not an authenticator extension, then  
0876 continue.  
0877  
0878 4. Let authenticatorExtensionInput be the (CBOR) result of  
0879 running extensionId's client extension processing algorithm on  
0880 clientExtensionInput. If the algorithm returned an error,  
0881 continue.  
0882  
0883 5. Set authenticatorExtensions[extensionId] to the base64url  
0884 encoding of authenticatorExtensionInput.  
0885  
0886 15. Let collectedClientData be a new CollectedClientData instance whose  
0887 fields are:  
0888  
0889 challenge  
0890 The base64url encoding of options.challenge.  
0891  
0892 origin  
0893 The serialization of callerOrigin.  
0894  
0895 hashAlgorithm  
0896 The recognized algorithm name of the hash algorithm  
0897 selected by the client for generating the hash of the  
0898 serialized client data.  
0899  
0900 tokenBindingId  
0901 The Token Binding ID associated with callerOrigin, if one  
0902 is available.  
0903  
0904 clientExtensions  
0905 clientExtensions  
0906  
0907 authenticatorExtensions  
0908 authenticatorExtensions

0862  
0863 16. Let clientDataJSON be the JSON-serialized client data constructed  
0864 from collectedClientData.  
0865 17. Let clientDataHash be the hash of the serialized client data  
0866 represented by clientDataJSON.  
0867 18. Let currentlyAvailableAuthenticators be a new ordered set  
0868 consisting of all authenticators currently available on this  
0869 platform.  
0870 19. Let selectedAuthenticators be a new ordered set.  
0871 20. If currentlyAvailableAuthenticators is empty, return a DOMException  
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  
0908 If the adjustedTimeout timer expires,  
0909 For each authenticator in issuedRequests invoke the  
0910 authenticatorCancel operation on authenticator and remove  
0911 authenticator from issuedRequests.  
0912  
0913 If any authenticator returns a status indicating that the user  
0914 cancelled the operation,  
0915  
0916 1. Remove authenticator from issuedRequests.  
0917 2. For each remaining authenticator in issuedRequests invoke  
0918 the authenticatorCancel operation on authenticator and  
0919 remove it from issuedRequests.  
0920  
0921 If any authenticator returns an error status,  
0922 Remove authenticator from issuedRequests.  
0923  
0924 If any authenticator indicates success,  
0925  
0926 1. Remove authenticator from issuedRequests.  
0927 2. Let attestationObject be a new ArrayBuffer, created using

0893  
0894 16. Let clientDataJSON be the JSON-serialized client data constructed  
0895 from collectedClientData.  
0896 17. Let clientDataHash be the hash of the serialized client data  
0897 represented by clientDataJSON.  
0898 18. Let currentlyAvailableAuthenticators be a new ordered set  
0899 consisting of all authenticators currently available on this  
0900 platform.  
0901 19. Let selectedAuthenticators be a new ordered set.  
0902 20. If currentlyAvailableAuthenticators is empty, return a DOMException  
0903 whose name is "NotFoundError", and terminate this algorithm.  
0904 21. If options.authenticatorSelection is present, iterate through  
0905 currentlyAvailableAuthenticators and do the following for each  
0906 authenticator:  
0907 1. If authenticatorAttachment is present and its value is not  
0908 equal to authenticator's attachment modality, continue.  
0909 2. If requireResidentKey is set to true and the authenticator is  
0910 not capable of storing a Client-Side-Resident Credential  
0911 Private Key, continue.  
0912 3. If requireUserVerification is set to true and the  
0913 authenticator is not capable of performing user verification,  
0914 continue.  
  
0915 4. Append authenticator to selectedAuthenticators.  
0916 22. If selectedAuthenticators is empty, return a DOMException whose  
0917 name is "ConstraintError", and terminate this algorithm.  
0918 23. Let issuedRequests be a new ordered set.  
0919 24. For each authenticator in currentlyAvailableAuthenticators:  
0920 1. Let excludeCredentialDescriptorList be a new list.  
0921 2. For each credential descriptor C in  
0922 options.excludeCredentials:  
0923 1. If C.transports is not empty, and authenticator is  
0924 connected over a transport not mentioned in C.transports,  
0925 the client MAY continue.  
0926 2. Otherwise, Append C to excludeCredentialDescriptorList.  
0927 3. In parallel, invoke the authenticatorMakeCredential operation  
0928 on authenticator with rpId, clientDataHash, options.rp,  
0929 options.user,  
0930 options.authenticatorSelection.requireResidentKey,  
0931 credTypesAndPubKeyAlgs, excludeCredentialDescriptorList, and  
0932 authenticatorExtensions as parameters.  
0933 4. Append authenticator to issuedRequests.  
0934 25. Start a timer for adjustedTimeout milliseconds. Then execute the  
0935 following steps in parallel. The task source for these tasks is the  
0936 dom manipulation task source.  
0937 26. While issuedRequests is not empty, perform the following actions  
0938 depending upon the adjustedTimeout timer and responses from the  
0939 authenticators:  
0940  
0941 If the adjustedTimeout timer expires,  
0942 For each authenticator in issuedRequests invoke the  
0943 authenticatorCancel operation on authenticator and remove  
0944 authenticator from issuedRequests.  
0945  
0946 If any authenticator returns a status indicating that the user  
0947 cancelled the operation,  
0948  
0949 1. Remove authenticator from issuedRequests.  
0950 2. For each remaining authenticator in issuedRequests invoke  
0951 the authenticatorCancel operation on authenticator and  
0952 remove it from issuedRequests.  
0953  
0954 If any authenticator returns an error status,  
0955 Remove authenticator from issuedRequests.  
0956  
0957 If any authenticator indicates success,  
0958  
0959 1. Remove authenticator from issuedRequests.  
0960 2. Let attestationObject be a new ArrayBuffer, created using



0928 global's %ArrayBuffer%, containing the bytes of the value  
 0929 returned from the successful authenticatorMakeCredential  
 0930 operation (which is attObj, as defined in 5.3.4  
 0931 Generating an Attestation Object).  
 0932 3. Let id be attestationObject.authData.attestation  
 0933 data.credential ID (see 5.3.1 Attestation data and 5.1  
 0934 Authenticator data).  
 0935 4. Let value be a new PublicKeyCredential object associated  
 0936 with global whose fields are:

0937     [[identifier]]  
 0938     id  
 0939  
 0940     response  
 0941     A new AuthenticatorAttestationResponse object  
 0942     associated with global whose fields are:

0943     clientDataJSON  
 0944     A new ArrayBuffer, created using  
 0945     global's %ArrayBuffer%, containing the  
 0946     bytes of clientDataJSON.  
 0947  
 0948     attestationObject  
 0949     attestationObject  
 0950  
 0951     clientExtensionResults  
 0952     A new AuthenticationExtensions object  
 0953     containing the extension identifier -> client  
 0954     extension output entries created by running  
 0955     each extension's client extension processing  
 0956     algorithm to create the client extension  
 0957     outputs, for each client extension in  
 0958     clientDataJSON.clientExtensions.

0959 5. For each remaining authenticator in issuedRequests invoke  
 0960 the authenticatorCancel operation on authenticator and  
 0961 remove it from issuedRequests.  
 0962 6. Return value and terminate this algorithm.

0963 27. Return a DOMException whose name is "NotAllowedError".

0964 During the above process, the user agent SHOULD show some UI to the  
 0965 user to guide them in the process of selecting and authorizing an  
 0966 authenticator.

0967 4.1.4. Use an existing credential to make an assertion -  
 0968 **PublicKeyCredential's [[DiscoverFromExternalSource]](options) method**  
 0969  
 0970 **The [[DiscoverFromExternalSource]](options) method is used to discover**  
 0971 **and use an existing public key credential, with the user's consent. The**  
 0972 **script optionally specifies some criteria to indicate what credentials**  
 0973 **are acceptable to it. The user agent and/or platform locates**  
 0974 **credentials matching the specified criteria, and guides the user to**  
 0975 **pick one that the script will be allowed to use. The user may choose**  
 0976 **not to provide a credential even if one is present, for example to**  
 0977 **maintain privacy.**

0978 **Note: This algorithm is synchronous; the Promise resolution/rejection**  
 0979 **is handled by navigator.credentials.get().**

0980  
 0981  
 0982  
 0983  
 0984  
 0985  
 0986  
 0987  
 0988 **This method accepts a single argument:**

0961 global's %ArrayBuffer%, containing the bytes of the value  
 0962 returned from the successful authenticatorMakeCredential  
 0963 operation (which is attObj, as defined in 6.3.4  
 0964 Generating an Attestation Object).  
 0965 3. Let id be attestationObject.authData.attestation  
 0966 data.credential ID (see 6.3.1 Attestation data and 6.1  
 0967 Authenticator data).  
 0968 4. Let value be a new PublicKeyCredential object associated  
 0969 with global whose fields are:

0970     [[identifier]]  
 0971     id  
 0972  
 0973     response  
 0974     A new AuthenticatorAttestationResponse object  
 0975     associated with global whose fields are:

0976     clientDataJSON  
 0977     A new ArrayBuffer, created using  
 0978     global's %ArrayBuffer%, containing the  
 0979     bytes of clientDataJSON.  
 0980  
 0981     attestationObject  
 0982     attestationObject  
 0983  
 0984     clientExtensionResults  
 0985     A new AuthenticationExtensions object  
 0986     containing the extension identifier -> client  
 0987     extension output entries created by running  
 0988     each extension's client extension processing  
 0989     algorithm to create the client extension  
 0990     outputs, for each client extension in  
 0991     clientDataJSON.clientExtensions.

0992 5. For each remaining authenticator in issuedRequests invoke  
 0993 the authenticatorCancel operation on authenticator and  
 0994 remove it from issuedRequests.  
 0995 6. Return value and terminate this algorithm.

0996 27. Return a DOMException whose name is "NotAllowedError".

0997 During the above process, the user agent SHOULD show some UI to the  
 0998 user to guide them in the process of selecting and authorizing an  
 0999 authenticator.

1000 5.1.4. Use an existing credential to make an assertion  
 1001  
 1002 **Relying Parties call navigator.credentials.get({publicKey:..., ...}) to**  
 1003 **discover and use an existing public key credential, with the user's**  
 1004 **consent. The script optionally specifies some criteria to indicate what**  
 1005 **credential sources are acceptable to it. The user agent and/or platform**  
 1006 **locates credential sources matching the specified criteria, and guides**  
 1007 **the user to pick one that the script will be allowed to use. The user**  
 1008 **may choose to decline the entire interaction even if a credential**  
 1009 **source is present, for example to maintain privacy. If the user picks a**  
 1010 **credential source, the user agent then uses 6.2.2 The**  
 1011 **authenticatorGetAssertion operation to sign a Relying Party-provided**  
 1012 **challenge and other collected data into an assertion, which is used as**  
 1013 **a credential.**

1014 **The get() implementation [CREDENTIAL-MANAGEMENT-1] calls**  
 1015 **PublicKeyCredential.[[CollectFromCredentialStore]]() to collect any**  
 1016 **credentials that should be available without user mediation (roughly,**  
 1017 **this specification's authorization gesture), and if it doesn't find**  
 1018 **exactly one of those, it calls**  
 1019 **PublicKeyCredential.[[DiscoverFromExternalSource]]() to have the user**  
 1020 **select a credential source.**

1021 **Since this specification requires an authorization gesture to create**

0985  
 0990 **options**  
 0991 This argument is a **CredentialRequestOptions** object whose  
 0992 **options.publicKey** member contains a challenge and additional  
 0993 options as described in 4.5 Options for Assertion Generation  
 0994 (dictionary **PublicKeyCredentialRequestOptions**). The selected  
 0995 authenticator signs the challenge along with other collected  
 0996 data in order to produce an assertion. See 5.2.2 The  
 0997 authenticatorGetAssertion operation.  
 0998  
 0999 When this method is invoked, the user agent MUST execute the following  
 1000 algorithm:  
 1001 1. Assert: **options.publicKey** is present.  
 1002 2. Let **options** be the value of **options.publicKey**.  
 1003 3. If the **timeout** member of **options** is present, check if its value  
 1004 lies within a reasonable range as defined by the platform and if  
 1005 not, correct it to the closest value lying within that range. Set  
 1006 **adjustedTimeout** to this adjusted value. If the **timeout** member of  
 1007 **options** is not present, then set **adjustedTimeout** to a  
 1008 platform-specific default.  
 1009 4. Let **global** be the **PublicKeyCredential**'s interface object's  
 1010 environment settings object's global object.  
 1011 5. Let **callerOrigin** be the origin specified by this  
 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 **effectiveDomain** 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.rpld** is not present, then set **rpld** to **effectiveDomain**.  
 1023 Otherwise:  
 1024 1. If **options.rpld** 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 **rpld** to **options.rpld**.  
 1028 Note: **rpld** 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.rpld** 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  
 1051 **origin**  
 1052 The serialization of **callerOrigin**.  
 1053  
 1054

1030 any credentials,  
 1031 **PublicKeyCredential.[[CollectFromCredentialStore]](options)** inherits  
 1032 the default behavior of **Credential.[[CollectFromCredentialStore]]()**, of  
 1033 returning an empty set.  
 1034  
 1035 5.1.4.1. **PublicKeyCredential**'s **[[DiscoverFromExternalSource]](options)**  
 1036 method  
 1037  
 1038 When the **PublicKeyCredential.[[DiscoverFromExternalSource]](options)**  
 1039 method is invoked, the user agent MUST:  
 1040 1. Assert: **options.publicKey** is present.  
 1041 2. Let **options** be the value of **options.publicKey**.  
 1042 3. If the **timeout** member of **options** is present, check if its value  
 1043 lies within a reasonable range as defined by the platform and if  
 1044 not, correct it to the closest value lying within that range. Set  
 1045 **adjustedTimeout** to this adjusted value. If the **timeout** member of  
 1046 **options** is not present, then set **adjustedTimeout** to a  
 1047 platform-specific default.  
 1048 4. Let **global** be the **PublicKeyCredential**'s interface object's  
 1049 environment settings object's global object.  
 1050 5. Let **callerOrigin** be the origin specified by this  
 1051 **PublicKeyCredential** interface object's relevant settings object. If  
 1052 **callerOrigin** is an opaque origin, return a **DOMException** whose name  
 1053 is "NotAllowedError", and terminate this algorithm.  
 1054 6. Let **effectiveDomain** be the **callerOrigin**'s effective domain. If  
 1055 **effectiveDomain** is not a valid domain, then return a **DOMException**  
 1056 whose name is "SecurityError" and terminate this algorithm.  
 1057 Note: An effective domain may resolve to a host, which can be  
 1058 represented in various manners, such as domain, ipv4 address, ipv6  
 1059 address, opaque host, or empty host. Only the domain format of host  
 1060 is allowed here.  
 1061 7. If **options.rpld** is not present, then set **rpld** to **effectiveDomain**.  
 1062 Otherwise:  
 1063 1. If **options.rpld** is not a registrable domain suffix of and is  
 1064 not equal to **effectiveDomain**, return a **DOMException** whose name  
 1065 is "SecurityError", and terminate this algorithm.  
 1066 2. Set **rpld** to **options.rpld**.  
 1067 Note: **rpld** represents the caller's RP ID. The RP ID defaults  
 1068 to being the caller's origin's effective domain unless the  
 1069 caller has explicitly set **options.rpld** when calling **get()**.  
 1070 8. Let **clientExtensions** be a new map and let **authenticatorExtensions**  
 1071 be a new map.  
 1072 9. If the **extensions** member of **options** is present, then for each  
 1073 **extensionId** -> **clientExtensionInput** of **options.extensions**:  
 1074 1. If **extensionId** is not supported by this client platform or is  
 1075 not an authentication extension, then continue.  
 1076 2. Set **clientExtensions[extensionId]** to **clientExtensionInput**.  
 1077 3. If **extensionId** is not an authenticator extension, then  
 1078 continue.  
 1079 4. Let **authenticatorExtensionInput** be the (CBOR) result of  
 1080 running **extensionId**'s client extension processing algorithm on  
 1081 **clientExtensionInput**. If the algorithm returned an error,  
 1082 continue.  
 1083 5. Set **authenticatorExtensions[extensionId]** to the base64url  
 1084 encoding of **authenticatorExtensionInput**.  
 1085 10. Let **collectedClientData** be a new **CollectedClientData** instance whose  
 1086 fields are:  
 1087 **challenge**  
 1088 The base64url encoding of **options.challenge**  
 1089  
 1090 **origin**  
 1091 The serialization of **callerOrigin**.  
 1092  
 1093

105E hashAlgorithm  
 105F The recognized algorithm name of the hash algorithm  
 1057 selected by the client for generating the hash of the  
 105E serialized client data

106C tokenBindingId  
 1061 The Token Binding ID associated with callerOrigin, if one  
 1062 is available.  
 1063

1064 clientExtensions  
 1065 clientExtensions  
 1066

1067 authenticatorExtensions  
 1068 authenticatorExtensions  
 1069

1070 11. Let clientDataJSON be the JSON-serialized client data constructed  
 1071 from collectedClientData.  
 1072 12. Let clientDataHash be the hash of the serialized client data  
 1073 represented by clientDataJSON.  
 1074 13. Let issuedRequests be a new ordered set.  
 1075 14. If there are no authenticators currently available on this  
 1076 platform, return a DOMException whose name is "NotFoundError", and  
 1077 terminate this algorithm.  
 1078 15. Let authenticator be a platform-specific handle whose value  
 1079 identifies an authenticator.  
 1080 16. For each authenticator currently available on this platform,  
 1081 perform the following steps:  
 1082 1. Let allowCredentialDescriptorList be a new list.  
 1083 2. If options.allowCredentials is not empty, execute a  
 1084 platform-specific procedure to determine which, if any, public  
 1085 key credentials described by options.allowCredentials are  
 1086 bound to this authenticator, by matching with rpId,  
 1087 options.allowCredentials.id, and  
 1088 options.allowCredentials.type. Set  
 1089 allowCredentialDescriptorList to this filtered list.  
 1090 3. If allowCredentialDescriptorList  
 1091 is not empty  
 1092  
 1093 1. Let distinctTransports be a new ordered set.  
 1094 2. For each credential descriptor C in  
 1095 allowCredentialDescriptorList, append each value, if  
 1096 any, of C.transports to distinctTransports.  
 1097 Note: This will aggregate only distinct values of  
 1098 transports (for this authenticator) in  
 1099 distinctTransports due to the properties of ordered  
 1100 sets.  
 1101 3. If distinctTransports  
 1102 is not empty  
 1103 The client selects one transport value  
 1104 from distinctTransports, possibly  
 1105 incorporating local configuration  
 1106 knowledge of the appropriate transport  
 1107 to use with authenticator in making its  
 1108 selection.  
 1109 Then, using transport, invoke in  
 1110 parallel the authenticatorGetAssertion  
 1111 operation on authenticator, with rpId,  
 1112 clientDataHash,  
 1113 allowCredentialDescriptorList, and  
 1114 authenticatorExtensions as parameters.  
 1115 is empty  
 1116 Using local configuration knowledge of  
 1117 the appropriate transport to use with  
 1118 authenticator, invoke in parallel the  
 1119 authenticatorGetAssertion operation on  
 1120 authenticator with rpId, clientDataHash,  
 1121  
 1122  
 1123  
 1124

1094 hashAlgorithm  
 1095 The recognized algorithm name of the hash algorithm  
 1096 selected by the client for generating the hash of the  
 1097 serialized client data

1098 tokenBindingId  
 1099 The Token Binding ID associated with callerOrigin, if one  
 1100 is available.  
 1101

1102 clientExtensions  
 1103 clientExtensions  
 1104

1105 authenticatorExtensions  
 1106 authenticatorExtensions  
 1107

1108 11. Let clientDataJSON be the JSON-serialized client data constructed  
 1109 from collectedClientData.  
 1110 12. Let clientDataHash be the hash of the serialized client data  
 1111 represented by clientDataJSON.  
 1112 13. Let issuedRequests be a new ordered set.  
 1113 14. If there are no authenticators currently available on this  
 1114 platform, return a DOMException whose name is "NotFoundError", and  
 1115 terminate this algorithm.  
 1116 15. Let authenticator be a platform-specific handle whose value  
 1117 identifies an authenticator.  
 1118 16. For each authenticator currently available on this platform,  
 1119 perform the following steps:  
 1120 1. Let allowCredentialDescriptorList be a new list.  
 1121 2. If options.allowCredentials is not empty, execute a  
 1122 platform-specific procedure to determine which, if any, public  
 1123 key credentials described by options.allowCredentials are  
 1124 bound to this authenticator, by matching with rpId,  
 1125 options.allowCredentials.id, and  
 1126 options.allowCredentials.type. Set  
 1127 allowCredentialDescriptorList to this filtered list.  
 1128 3. If allowCredentialDescriptorList  
 1129 is not empty  
 1130  
 1131 1. Let distinctTransports be a new ordered set.  
 1132 2. For each credential descriptor C in  
 1133 allowCredentialDescriptorList, append each value, if  
 1134 any, of C.transports to distinctTransports.  
 1135 Note: This will aggregate only distinct values of  
 1136 transports (for this authenticator) in  
 1137 distinctTransports due to the properties of ordered  
 1138 sets.  
 1139 3. If distinctTransports  
 1140 is not empty  
 1141 The client selects one transport value  
 1142 from distinctTransports, possibly  
 1143 incorporating local configuration  
 1144 knowledge of the appropriate transport  
 1145 to use with authenticator in making its  
 1146 selection.  
 1147 Then, using transport, invoke in  
 1148 parallel the authenticatorGetAssertion  
 1149 operation on authenticator, with rpId,  
 1150 clientDataHash,  
 1151 allowCredentialDescriptorList, and  
 1152 authenticatorExtensions as parameters.  
 1153 is empty  
 1154 Using local configuration knowledge of  
 1155 the appropriate transport to use with  
 1156 authenticator, invoke in parallel the  
 1157 authenticatorGetAssertion operation on  
 1158 authenticator with rpId, clientDataHash,  
 1159  
 1160  
 1161  
 1162  
 1163



1125 allowCredentialDescriptorList, and  
 1126 clientExtensions as parameters.  
 1127  
 1128 is empty  
 1129 Using local configuration knowledge of the  
 1130 appropriate transport to use with authenticator,  
 1131 invoke in parallel the authenticatorGetAssertion  
 1132 operation on authenticator with rpId,  
 1133 clientDataHash, and clientExtensions as parameters.  
 1134  
 1135 Note: In this case, the Relying Party did not supply  
 1136 a list of acceptable credential descriptors. Thus  
 1137 the authenticator is being asked to exercise any  
 1138 credential it may possess that is bound to the  
 1139 Relying Party, as identified by rpId.  
 1140  
 1141 4. Append authenticator to issuedRequests.  
 1142 17. Start a timer for adjustedTimeout milliseconds. Then execute the  
 1143 following steps in parallel. The task source for these tasks is the  
 1144 dom manipulation task source.  
 1145 18. While issuedRequests is not empty, perform the following actions  
 1146 depending upon the adjustedTimeout timer and responses from the  
 1147 authenticators:  
 1148  
 1149 If the adjustedTimeout timer expires,  
 1150 For each authenticator in issuedRequests invoke the  
 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

1164 allowCredentialDescriptorList, and  
 1165 clientExtensions as parameters.  
 1166  
 1167 is empty  
 1168 Using local configuration knowledge of the  
 1169 appropriate transport to use with authenticator,  
 1170 invoke in parallel the authenticatorGetAssertion  
 1171 operation on authenticator with rpId,  
 1172 clientDataHash, and clientExtensions as parameters.  
 1173  
 1174 Note: In this case, the Relying Party did not supply  
 1175 a list of acceptable credential descriptors. Thus  
 1176 the authenticator is being asked to exercise any  
 1177 credential it may possess that is bound to the  
 1178 Relying Party, as identified by rpId.  
 1179  
 1180 4. Append authenticator to issuedRequests.  
 1181 17. Start a timer for adjustedTimeout milliseconds. Then execute the  
 1182 following steps in parallel. The task source for these tasks is the  
 1183 dom manipulation task source.  
 1184 18. While issuedRequests is not empty, perform the following actions  
 1185 depending upon the adjustedTimeout timer and responses from the  
 1186 authenticators:  
 1187  
 1188 If the adjustedTimeout timer expires,  
 1189 For each authenticator in issuedRequests invoke the  
 1190 authenticatorCancel operation on authenticator and remove  
 1191 authenticator from issuedRequests.  
 1192  
 1193 If any authenticator returns a status indicating that the user  
 1194 cancelled the operation,  
 1195  
 1196 1. Remove authenticator from issuedRequests.  
 1197 2. For each remaining authenticator in issuedRequests invoke  
 1198 the authenticatorCancel operation on authenticator and  
 1199 remove it from issuedRequests.  
 1200  
 1201 If any authenticator returns an error status,  
 1202 Remove authenticator from issuedRequests.  
 1203  
 1204 If any authenticator indicates success,  
 1205  
 1206 1. Remove authenticator from issuedRequests.  
 1207 2. Let value be a new PublicKeyCredential associated with  
 1208 global whose fields are:  
 1209  
 1210 [[identifier]]  
 1211 A new ArrayBuffer, created using global's  
 1212 %ArrayBuffer%, containing the bytes of the  
 1213 credential ID returned from the successful  
 1214 authenticatorGetAssertion operation, as  
 1215 defined in 6.2.2 The  
 1216 authenticatorGetAssertion operation.  
 1217  
 1218 response  
 1219 A new AuthenticatorAssertionResponse object  
 1220 associated with global whose fields are:  
 1221  
 1222 clientDataJSON  
 1223 A new ArrayBuffer, created using  
 1224 global's %ArrayBuffer%, containing the  
 1225 bytes of clientDataJSON  
 1226  
 1227 authenticatorData  
 1228 A new ArrayBuffer, created using  
 1229 global's %ArrayBuffer%, containing the  
 1230 bytes of the returned authenticatorData  
 1231  
 1232 signature  
 1233 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  
 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

1234 global's %ArrayBuffer%, containing the  
 1235 bytes of the returned signature  
 1236

1237 **userHandle**  
 1238 A new ArrayBuffer, created using  
 1239 global's %ArrayBuffer%, containing the  
 1240 user handle returned from the successful  
 1241 authenticatorGetAssertion operation, as  
 1242 defined in 6.2.2 The  
 1243 authenticatorGetAssertion operation.  
 1244

1245 **clientExtensionResults**  
 1246 A new AuthenticationExtensions object  
 1247 containing the extension identifier -> client  
 1248 extension output entries created by running  
 1249 each extension's client extension processing  
 1250 algorithm to create the client extension  
 1251 outputs, for each client extension in  
 1252 clientDataJSON.clientExtensions.  
 1253

1254 3. For each remaining authenticator in issuedRequests invoke  
 1255 the authenticatorCancel operation on authenticator and  
 1256 remove it from issuedRequests.  
 1257 4. Return value and terminate this algorithm.  
 1258

1259 19. Return a DOMException whose name is "NotAllowedError".  
 1260

1261 During the above process, the user agent SHOULD show some UI to the  
 1262 user to guide them in the process of selecting and authorizing an  
 1263 authenticator with which to complete the operation.  
 1264

1265 **5.1.5. Store an existing credential - PublicKeyCredential's**  
 1266 **[[Store]](credential) method**  
 1267  
 1268 The **[[Store]](credential)** method is not supported for Web  
 1269 Authentication's PublicKeyCredential type, so it always returns an  
 1270 error.  
 1271

1272 **Note:** This algorithm is synchronous; the Promise resolution/rejection  
 1273 is handled by navigator.credentials.store().  
 1274

1275 **This method accepts a single argument:**  
 1276

1277 **credential**  
 1278 This argument is a PublicKeyCredential object.  
 1279

1280 **When this method is invoked, the user agent MUST execute the following**  
 1281 **algorithm:**  
 1282 1. Return a DOMException whose name is "NotSupportedError", and  
 1283 terminate this algorithm  
 1284

1285 **5.1.6. Platform Authenticator Availability - PublicKeyCredential's**  
 1286 **isPlatformAuthenticatorAvailable() method**  
 1287  
 1288 Relying Parties use this method to determine whether they can create a  
 1289 new credential using a platform authenticator. Upon invocation, the  
 1290 client employs a platform-specific procedure to discover available  
 1291 platform authenticators. If successful, the client then assesses  
 1292 whether the user is willing to create a credential using one of the  
 1293 available platform authenticators. This assessment may include various  
 1294 factors, such as:  
 1295 \* Whether the user is running in private or incognito mode.  
 1296 \* Whether the user has configured the client to not create such  
 1297 credentials.  
 1298 \* Whether the user has previously expressed an unwillingness to  
 1299 create a new credential for this Relying Party, either through  
 1300 configuration or by declining a user interface prompt.  
 1301 \* The user's explicitly stated intentions, determined through user  
 1302 interaction.  
 1303

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  
 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

1304 If this assessment is affirmative, the promise is resolved with the  
 1305 value of True. Otherwise, the promise is resolved with the value of  
 1306 False. Based on the result, the Relying Party can take further actions  
 1307 to guide the user to create a credential.  
 1308  
 1309 This method has no arguments and returns a boolean value.  
 1310  
 1311 If the promise will return False, the client SHOULD wait a fixed period  
 1312 of time from the invocation of the method before returning False. This  
 1313 is done so that callers can not distinguish between the case where the  
 1314 user was unwilling to create a credential using one of the available  
 1315 platform authenticators and the case where no platform authenticator  
 1316 exists. Trying to make these cases indistinguishable is done in an  
 1317 attempt to not provide additional information that could be used for  
 1318 fingerprinting. A timeout value on the order of 10 minutes is  
 1319 recommended; this is enough time for successful user interactions to be  
 1320 performed but short enough that the dangling promise will still be  
 1321 resolved in a reasonably timely fashion.  
 1322 [SecureContext]  
 1323 partial interface PublicKeyCredential {  
 1324 static Promise < boolean > isPlatformAuthenticatorAvailable();  
 1325 };  
 1326  
 1327 **5.2. Authenticator Responses (interface AuthenticatorResponse)**  
 1328  
 1329 Authenticators respond to Relying Party requests by returning an object  
 1330 derived from the AuthenticatorResponse interface:  
 1331 [SecureContext]  
 1332 interface AuthenticatorResponse {  
 1333 [SameObject] readonly attribute ArrayBuffer clientDataJSON;  
 1334 };  
 1335  
 1336 clientDataJSON, of type ArrayBuffer, readonly  
 1337 This attribute contains a JSON serialization of the client data  
 1338 passed to the authenticator by the client in its call to either  
 1339 create() or get().  
 1340  
 1341 **5.2.1. Information about Public Key Credential (interface**  
 1342 **AuthenticatorAttestationResponse)**  
 1343  
 1344 The AuthenticatorAttestationResponse interface represents the  
 1345 authenticator's response to a client's request for the creation of a  
 1346 new public key credential. It contains information about the new  
 1347 credential that can be used to identify it for later use, and metadata  
 1348 that can be used by the Relying Party to assess the characteristics of  
 1349 the credential during registration.  
 1350 [SecureContext]  
 1351 interface AuthenticatorAttestationResponse : AuthenticatorResponse {  
 1352 [SameObject] readonly attribute ArrayBuffer attestationObject;  
 1353 };  
 1354  
 1355 clientDataJSON  
 1356 This attribute, inherited from AuthenticatorResponse, contains  
 1357 the JSON-serialized client data (see 6.3 Attestation) passed to  
 1358 the authenticator by the client in order to generate this  
 1359 credential. The exact JSON serialization must be preserved, as  
 1360 the hash of the serialized client data has been computed over  
 1361 it.  
 1362  
 1363 attestationObject, of type ArrayBuffer, readonly  
 1364 This attribute contains an attestation object, which is opaque  
 1365 to, and cryptographically protected against tampering by, the  
 1366 client. The attestation object contains both authenticator data  
 1367 and an attestation statement. The former contains the AAGUID, a  
 1368 unique credential ID, and the credential public key. The  
 1369 contents of the attestation statement are determined by the  
 1370 attestation statement format used by the authenticator. It also  
 1371 contains any additional information that the Relying Party's  
 1372 server requires to validate the attestation statement, as well  
 1373 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  
 1346 PublicKeyCredentialParameters)  
 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.  
 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;

1374 JSON-serialized client data. For more details, see 6.3  
 1375 Attestation, 6.3.4 Generating an Attestation Object, and Figure  
 1376 3.  
 1377  
 1378 5.2.2. Web Authentication Assertion (interface  
 1379 AuthenticatorAssertionResponse)  
 1380  
 1381 The AuthenticatorAssertionResponse interface represents an  
 1382 authenticator's response to a client's request for generation of a new  
 1383 authentication assertion given the Relying Party's challenge and  
 1384 optional list of credentials it is aware of. This response contains a  
 1385 cryptographic signature proving possession of the credential private  
 1386 key, and optionally evidence of user consent to a specific transaction.  
 1387 [SecureContext]  
 1388 interface AuthenticatorAssertionResponse : AuthenticatorResponse {  
 1389 [SameObject] readonly attribute ArrayBuffer authenticatorData;  
 1390 [SameObject] readonly attribute ArrayBuffer signature;  
 1391 [SameObject] readonly attribute ArrayBuffer userHandle;  
 1392 };  
 1393  
 1394 clientDataJSON  
 1395 This attribute, inherited from AuthenticatorResponse, contains  
 1396 the JSON-serialized client data (see 5.7.1 Client data used in  
 1397 WebAuthn signatures (dictionary CollectedClientData)) passed to  
 1398 the authenticator by the client in order to generate this  
 1399 assertion. The exact JSON serialization must be preserved, as  
 1400 the hash of the serialized client data has been computed over  
 1401 it.  
 1402  
 1403 authenticatorData, of type ArrayBuffer, readonly  
 1404 This attribute contains the authenticator data returned by the  
 1405 authenticator. See 6.1 Authenticator data.  
 1406  
 1407 signature, of type ArrayBuffer, readonly  
 1408 This attribute contains the raw signature returned from the  
 1409 authenticator. See 6.2.2 The authenticatorGetAssertion  
 1410 operation.  
 1411  
 1412 userHandle, of type ArrayBuffer, readonly  
 1413 This attribute contains the user handle returned from the  
 1414 authenticator. See 6.2.2 The authenticatorGetAssertion  
 1415 operation.  
 1416  
 1417 5.3. Parameters for Credential Generation (dictionary  
 1418 PublicKeyCredentialParameters)  
 1419  
 1420 dictionary PublicKeyCredentialParameters {  
 1421 required PublicKeyCredentialType type;  
 1422 required COSEAlgorithmIdentifier alg;  
 1423 };  
 1424  
 1425 This dictionary is used to supply additional parameters when creating a  
 1426 new credential.  
 1427  
 1428 The type member specifies the type of credential to be created.  
 1429  
 1430 The alg member specifies the cryptographic signature algorithm with  
 1431 which the newly generated credential will be used, and thus also the  
 1432 type of asymmetric key pair to be generated, e.g., RSA or Elliptic  
 1433 Curve.  
 1434  
 1435 Note: we use "alg" as the latter member name, rather than spelling-out  
 1436 "algorithm", because it will be serialized into a message to the  
 1437 authenticator, which may be sent over a low-bandwidth link.  
 1438  
 1439 5.4. Options for Credential Creation (dictionary  
 1440 MakePublicKeyCredentialOptions)  
 1441  
 1442 dictionary MakePublicKeyCredentialOptions {  
 1443 required PublicKeyCredentialRpEntity 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
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.

```

```

1444 required PublicKeyCredentialUserEntity user;
1445
1446 required BufferSource challenge;
1447 required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
1448
1449 unsigned long timeout;
1450 sequence<PublicKeyCredentialDescriptor> excludeCredentials = [];
1451 AuthenticatorSelectionCriteria authenticatorSelection;
1452 AuthenticationExtensions extensions;
1453 };
1454
1455 rp, of type PublicKeyCredentialRpEntity
1456 This member contains data about the Relying Party responsible
1457 for the request.
1458
1459 Its value's name member is required, and contains the friendly
1460 name of the Relying Party (e.g. "Acme Corporation", "Widgets,
1461 Inc.", or "Awesome Site").
1462
1463 Its value's id member specifies the relying party identifier
1464 with which the credential should be associated. If omitted, its
1465 value will be the CredentialsContainer object's relevant
1466 settings object's origin's effective domain.
1467
1468 user, of type PublicKeyCredentialUserEntity
1469 This member contains data about the user account for which the
1470 Relying Party is requesting attestation.
1471
1472 Its value's name member is required, and contains a name for the
1473 user account (e.g., "john.p.smith@example.com" or
1474 "+14255551234").
1475
1476 Its value's displayName member is required, and contains a
1477 friendly name for the user account (e.g., "John P. Smith").
1478
1479 Its value's id member is required and contains the user handle
1480 for the account, specified by the Relying Party.
1481
1482 challenge, of type BufferSource
1483 This member contains a challenge intended to be used for
1484 generating the newly created credential's attestation object.
1485
1486 pubKeyCredParams, of type sequence<PublicKeyCredentialParameters>
1487 This member contains information about the desired properties of
1488 the credential to be created. The sequence is ordered from most
1489 preferred to least preferred. The platform makes a best-effort
1490 to create the most preferred credential that it can.
1491
1492 timeout, of type unsigned long
1493 This member specifies a time, in milliseconds, that the caller
1494 is willing to wait for the call to complete. This is treated as
1495 a hint, and may be overridden by the platform.
1496
1497 excludeCredentials, of type sequence<PublicKeyCredentialDescriptor>,
1498 defaulting to None
1499 This member is intended for use by Relying Parties that wish to
1500 limit the creation of multiple credentials for the same account
1501 on a single authenticator. The platform is requested to return
1502 an error if the new credential would be created on an
1503 authenticator that also contains one of the credentials
1504 enumerated in this parameter.
1505
1506 authenticatorSelection, of type AuthenticatorSelectionCriteria
1507 This member is intended for use by Relying Parties that wish to
1508 select the appropriate authenticators to participate in the
1509 create() or get() operation.

```

1441 extensions, of type AuthenticationExtensions  
 1442 This member contains additional parameters requesting additional  
 1443 processing by the client and authenticator. For example, the  
 1444 caller may request that only authenticators with certain  
 1445 capabilities be used to create the credential, or that particular  
 1446 information be returned in the attestation object. Some  
 1447 extensions are defined in 8 WebAuthn Extensions; consult the  
 1448 IANA "WebAuthn Extension Identifier" registry established by  
 1449 [WebAuthn-Registries] for an up-to-date list of registered  
 1450 WebAuthn Extensions.  
 1451  
 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**  
 1491 **AuthenticatorSelectionCriteria)**  
 1492  
 1493 Relying Parties may use the AuthenticatorSelectionCriteria dictionary

1510 extensions, of type AuthenticationExtensions  
 1511 This member contains additional parameters requesting additional  
 1512 processing by the client and authenticator. For example, the  
 1513 caller may request that only authenticators with certain  
 1514 capabilities be used to create the credential, or that particular  
 1515 information be returned in the attestation object. Some  
 1516 extensions are defined in 9 WebAuthn Extensions; consult the  
 1517 IANA "WebAuthn Extension Identifier" registry established by  
 1518 [WebAuthn-Registries] for an up-to-date list of registered  
 1519 WebAuthn Extensions.  
 1520  
 1521  
 1522 **5.4.1. Public Key Entity Description (dictionary PublicKeyCredentialEntity)**  
 1523  
 1524 The PublicKeyCredentialEntity dictionary describes a user account, or a  
 1525 Relying Party, with which a public key credential is associated.  
 1526 dictionary PublicKeyCredentialEntity {  
 1527 **DOMString name;**  
 1528 **USVString icon;**  
 1529 };  
 1530  
 1531  
 1532 **name, of type DOMString**  
 1533 **A human-friendly identifier for the entity. For example, this**  
 1534 **could be a company name for a Relying Party, or a user's name.**  
 1535 **This identifier is intended for display.**  
 1536  
 1537 **icon, of type USVString**  
 1538 **A serialized URL which resolves to an image associated with the**  
 1539 **entity. For example, this could be a user's avatar or a Relying**  
 1540 **Party's logo. This URL MUST be an a priori authenticated URL.**  
 1541  
 1542 **5.4.2. RP Parameters for Credential Generation (dictionary**  
 1543 **PublicKeyCredentialRpEntity)**  
 1544  
 1545 The PublicKeyCredentialRpEntity dictionary is used to supply additional  
 1546 Relying Party attributes when creating a new credential.  
 1547 dictionary PublicKeyCredentialRpEntity : PublicKeyCredentialEntity {  
 1548 **DOMString id;**  
 1549 };  
 1550  
 1551 **id, of type DOMString**  
 1552 **A unique identifier for the Relying Party entity, which sets the**  
 1553 **RP ID.**  
 1554  
 1555 **5.4.3. User Account Parameters for Credential Generation (dictionary**  
 1556 **PublicKeyCredentialUserEntity)**  
 1557  
 1558 The PublicKeyCredentialUserEntity dictionary is used to supply  
 1559 additional user account attributes when creating a new credential.  
 1560 dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {  
 1561 **BufferSource id;**  
 1562 **DOMString displayName;**  
 1563 };  
 1564  
 1565 **id, of type BufferSource**  
 1566 **The user handle of the user account entity.**  
 1567  
 1568 **displayName, of type DOMString**  
 1569 **A friendly name for the user account (e.g., "John P. Smith").**  
 1570  
 1571 **5.4.4. Authenticator Selection Criteria (dictionary**  
 1572 **AuthenticatorSelectionCriteria)**  
 1573  
 1574 Relying Parties may use the AuthenticatorSelectionCriteria dictionary



```

1494 to specify their requirements regarding authenticator attributes.
1495 dictionary AuthenticatorSelectionCriteria {
1496   AuthenticatorAttachment aa; // authenticatorAttachment
1497   boolean rk = false; // requireResidentKey
1498   boolean uv = false; // requireUserVerification
1499 };
1500
1501 aa (AuthenticatorAttachment), of type AuthenticatorAttachment
1502 If this member is present, eligible authenticators are filtered
1503 to only authenticators attached with the specified 4.4.4
1504 Authenticator Attachment enumeration (enum
1505 AuthenticatorAttachment).
1506
1507 rk (requireResidentKey), of type boolean, defaulting to false
1508 This member describes the Relying Parties' requirements
1509 regarding availability of the Client-side-resident Credential
1510 Private Key. If the parameter is set to true, the authenticator
1511 MUST create a Client-side-resident Credential Private Key when
1512 creating a public key credential.
1513
1514 uv (requireUserVerification), of type boolean, defaulting to false
1515 This member describes the Relying Parties' requirements
1516 regarding the authenticator being capable of performing user
1517 verification. If the parameter is set to true, the authenticator
1518 MUST perform user verification when performing the create()
1519 operation and future 4.1.4 Use an existing credential to make
1520 an assertion - PublicKeyCredential's
1521 [[DiscoverFromExternalSource]](options) method operations when
1522 it is requested to verify the credential.
1523
1524 Note: These identifiers are intentionally short, rather than
1525 descriptive, because they will be serialized into a message to the
1526 authenticator, which may be sent over a low-bandwidth link.
1527
1528 4.4.4. Authenticator Attachment enumeration (enum AuthenticatorAttachment)
1529
1530 enum AuthenticatorAttachment {
1531   "plat", // Platform attachment
1532   "xplat" // Cross-platform attachment
1533 };
1534
1535 Clients may communicate with authenticators using a variety of
1536 mechanisms. For example, a client may use a platform-specific API to
1537 communicate with an authenticator which is physically bound to a
1538 platform. On the other hand, a client may use a variety of standardized
1539 cross-platform transport protocols such as Bluetooth (see 4.7.4
1540 Authenticator Transport enumeration (enum AuthenticatorTransport)) to
1541 discover and communicate with cross-platform attached authenticators.
1542 Therefore, we use AuthenticatorAttachment to describe an
1543 authenticator's attachment modality. We define authenticators that are
1544 part of the client's platform as having a platform attachment, and
1545 refer to them as platform authenticators. While those that are
1546 reachable via cross-platform transport protocols are defined as having
1547 cross-platform attachment, and refer to them as roaming authenticators.
1548 * platform attachment - the respective authenticator is attached
1549 using platform-specific transports. Usually, authenticators of this
1550 class are non-removable from the platform.
1551 * cross-platform attachment - the respective authenticator is
1552 attached using cross-platform transports. Authenticators of this
1553 class are removable from, and can "roam" among, client platforms.
1554
1555 This distinction is important because there are use-cases where only
1556 platform authenticators are acceptable to a Relying Party, and
1557 conversely ones where only roaming authenticators are employed. As a
1558 concrete example of the former, a credential on a platform
1559 authenticator may be used by Relying Parties to quickly and
1560 conveniently reauthenticate the user with a minimum of friction, e.g.,
1561 the user will not have to dig around in their pocket for their key fob
1562 or phone. As a concrete example of the latter, when the user is
1563 accessing the Relying Party from a given client for the first time,

```

```

1574 to specify their requirements regarding authenticator attributes.
1575 dictionary AuthenticatorSelectionCriteria {
1576   AuthenticatorAttachment authenticatorAttachment;
1577   boolean requireResidentKey = false;
1578   boolean requireUserVerification = false;
1579 };
1580
1581 authenticatorAttachment, of type AuthenticatorAttachment
1582 If this member is present, eligible authenticators are filtered
1583 to only authenticators attached with the specified 5.4.5
1584 Authenticator Attachment enumeration (enum
1585 AuthenticatorAttachment).
1586
1587 requireResidentKey, of type boolean, defaulting to false
1588 This member describes the Relying Parties' requirements
1589 regarding availability of the Client-side-resident Credential
1590 Private Key. If the parameter is set to true, the authenticator
1591 MUST create a Client-side-resident Credential Private Key when
1592 creating a public key credential.
1593
1594 requireUserVerification, of type boolean, defaulting to false
1595 This member describes the Relying Parties' requirements
1596 regarding the authenticator being capable of performing user
1597 verification. If the parameter is set to true, the authenticator
1598 MUST perform user verification when performing the create()
1599 operation and future 5.1.4 Use an existing credential to make
1600 an assertion operations when it is requested to verify the
1601 credential.
1602
1603 5.4.5. Authenticator Attachment enumeration (enum AuthenticatorAttachment)
1604
1605 enum AuthenticatorAttachment {
1606   "platform", // Platform attachment
1607   "cross-platform" // Cross-platform attachment
1608 };
1609
1610 Clients may communicate with authenticators using a variety of
1611 mechanisms. For example, a client may use a platform-specific API to
1612 communicate with an authenticator which is physically bound to a
1613 platform. On the other hand, a client may use a variety of standardized
1614 cross-platform transport protocols such as Bluetooth (see 5.7.4
1615 Authenticator Transport enumeration (enum AuthenticatorTransport)) to
1616 discover and communicate with cross-platform attached authenticators.
1617 Therefore, we use AuthenticatorAttachment to describe an
1618 authenticator's attachment modality. We define authenticators that are
1619 part of the client's platform as having a platform attachment, and
1620 refer to them as platform authenticators. While those that are
1621 reachable via cross-platform transport protocols are defined as having
1622 cross-platform attachment, and refer to them as roaming authenticators.
1623 * platform attachment - the respective authenticator is attached
1624 using platform-specific transports. Usually, authenticators of this
1625 class are non-removable from the platform.
1626 * cross-platform attachment - the respective authenticator is
1627 attached using cross-platform transports. Authenticators of this
1628 class are removable from, and can "roam" among, client platforms.
1629
1630 This distinction is important because there are use-cases where only
1631 platform authenticators are acceptable to a Relying Party, and
1632 conversely ones where only roaming authenticators are employed. As a
1633 concrete example of the former, a credential on a platform
1634 authenticator may be used by Relying Parties to quickly and
1635 conveniently reauthenticate the user with a minimum of friction, e.g.,
1636 the user will not have to dig around in their pocket for their key fob
1637 or phone. As a concrete example of the latter, when the user is
1638 accessing the Relying Party from a given client for the first time,

```

1564 they may be required to use a roaming authenticator which was  
 1565 originally registered with the Relying Party using a different client.  
 1566

1567 **4.5. Options for Assertion Generation (dictionary**  
 1568 **PublicKeyCredentialRequestOptions)**  
 1569

1570 The `PublicKeyCredentialRequestOptions` dictionary supplies `get()` with  
 1571 the data it needs to generate an assertion. Its challenge member must  
 1572 be present, while its other members are optional.  
 1573 dictionary `PublicKeyCredentialRequestOptions` {  
 1574 required `BufferSource` challenge;  
 1575 unsigned long timeout;  
 1576 `USVString` rpId;  
 1577 sequence<`PublicKeyCredentialDescriptor`> allowCredentials = [];  
 1578 `AuthenticationExtensions` extensions;  
 1579 };

1580 challenge, of type `BufferSource`  
 1581 This member represents a challenge that the selected  
 1582 authenticator signs, along with other data, when producing an  
 1583 authentication assertion.  
 1584

1585 timeout, of type unsigned long  
 1586 This optional member specifies a time, in milliseconds, that the  
 1587 caller is willing to wait for the call to complete. The value is  
 1588 treated as a hint, and may be overridden by the platform.  
 1589

1590 rpId, of type `USVString`  
 1591 This optional member specifies the relying party identifier  
 1592 claimed by the caller. If omitted, its value will be the  
 1593 `CredentialsContainer` object's relevant settings object's  
 1594 origin's effective domain.  
 1595

1596 allowCredentials, of type sequence<`PublicKeyCredentialDescriptor`>,  
 1597 defaulting to `None`  
 1598 This optional member contains a list of  
 1599 `PublicKeyCredentialDescriptor` object representing public key  
 1600 credentials acceptable to the caller, in decending order of the  
 1601 caller's preference (the first item in the list is the most  
 1602 preferred credential, and so on down the list).  
 1603

1604 extensions, of type `AuthenticationExtensions`  
 1605 This optional member contains additional parameters requesting  
 1606 additional processing by the client and authenticator. For  
 1607 example, if transaction confirmation is sought from the user,  
 1608 then the prompt string might be included as an extension.  
 1609

1610 **4.6. Authentication Extensions (typedef `AuthenticationExtensions`)**  
 1611

1612 typedef record<`DOMString`, any> `AuthenticationExtensions`;  
 1613

1614 This is a dictionary containing zero or more `WebAuthn` extensions, as  
 1615 defined in 8 `WebAuthn` Extensions. An `AuthenticationExtensions` instance  
 1616 can contain either client extensions or authenticator extensions,  
 1617 depending upon context.  
 1618

1619 **4.7. Supporting Data Structures**  
 1620

1621 The public key credential type uses certain data structures that are  
 1622 specified in supporting specifications. These are as follows.  
 1623

1624 **4.7.1. Client data used in `WebAuthn` signatures (dictionary**  
 1625 **`CollectedClientData`)**  
 1626

1627 The client data represents the contextual bindings of both the Relying  
 1628 Party and the client platform. It is a key-value mapping with  
 1629 string-valued keys. Values may be any type that has a valid encoding in  
 1630 JSON. Its structure is defined by the following Web IDL.  
 1631 dictionary `CollectedClientData` {  
 1632 required `DOMString` challenge;  
 1633

1639 they may be required to use a roaming authenticator which was  
 1640 originally registered with the Relying Party using a different client.  
 1641

1642 **5.5. Options for Assertion Generation (dictionary**  
 1643 **PublicKeyCredentialRequestOptions)**  
 1644

1645 The `PublicKeyCredentialRequestOptions` dictionary supplies `get()` with  
 1646 the data it needs to generate an assertion. Its challenge member must  
 1647 be present, while its other members are optional.  
 1648 dictionary `PublicKeyCredentialRequestOptions` {  
 1649 required `BufferSource` challenge;  
 1650 unsigned long timeout;  
 1651 `USVString` rpId;  
 1652 sequence<`PublicKeyCredentialDescriptor`> allowCredentials = [];  
 1653 `AuthenticationExtensions` extensions;  
 1654 };

1655 challenge, of type `BufferSource`  
 1656 This member represents a challenge that the selected  
 1657 authenticator signs, along with other data, when producing an  
 1658 authentication assertion.  
 1659

1660 timeout, of type unsigned long  
 1661 This optional member specifies a time, in milliseconds, that the  
 1662 caller is willing to wait for the call to complete. The value is  
 1663 treated as a hint, and may be overridden by the platform.  
 1664

1665 rpId, of type `USVString`  
 1666 This optional member specifies the relying party identifier  
 1667 claimed by the caller. If omitted, its value will be the  
 1668 `CredentialsContainer` object's relevant settings object's  
 1669 origin's effective domain.  
 1670

1671 allowCredentials, of type sequence<`PublicKeyCredentialDescriptor`>,  
 1672 defaulting to `None`  
 1673 This optional member contains a list of  
 1674 `PublicKeyCredentialDescriptor` object representing public key  
 1675 credentials acceptable to the caller, in decending order of the  
 1676 caller's preference (the first item in the list is the most  
 1677 preferred credential, and so on down the list).  
 1678

1679 extensions, of type `AuthenticationExtensions`  
 1680 This optional member contains additional parameters requesting  
 1681 additional processing by the client and authenticator. For  
 1682 example, if transaction confirmation is sought from the user,  
 1683 then the prompt string might be included as an extension.  
 1684

1685 **5.6. Authentication Extensions (typedef `AuthenticationExtensions`)**  
 1686

1687 typedef record<`DOMString`, any> `AuthenticationExtensions`;  
 1688

1689 This is a dictionary containing zero or more `WebAuthn` extensions, as  
 1690 defined in 9 `WebAuthn` Extensions. An `AuthenticationExtensions` instance  
 1691 can contain either client extensions or authenticator extensions,  
 1692 depending upon context.  
 1693

1694 **5.7. Supporting Data Structures**  
 1695

1696 The public key credential type uses certain data structures that are  
 1697 specified in supporting specifications. These are as follows.  
 1698

1699 **5.7.1. Client data used in `WebAuthn` signatures (dictionary**  
 1700 **`CollectedClientData`)**  
 1701

1702 The client data represents the contextual bindings of both the Relying  
 1703 Party and the client platform. It is a key-value mapping with  
 1704 string-valued keys. Values may be any type that has a valid encoding in  
 1705 JSON. Its structure is defined by the following Web IDL.  
 1706 dictionary `CollectedClientData` {  
 1707 required `DOMString` challenge;  
 1708

```

1634 required DOMString      origin;
1635 required DOMString      hashAlgorithm;
1636 DOMString                tokenBindingId;
1637 AuthenticationExtensions clientExtensions;
1638 AuthenticationExtensions authenticatorExtensions;
1639 };
1640
1641 The challenge member contains the base64url encoding of the challenge
1642 provided by the RP.
1643
1644 The origin member contains the fully qualified origin of the requester,
1645 as provided to the authenticator by the client, in the syntax defined
1646 by [RFC6454].
1647
1648 The hashAlgorithm member is a recognized algorithm name that supports
1649 the "digest" operation, which specifies the algorithm used to compute
1650 the hash of the serialized client data. This algorithm is chosen by the
1651 client at its sole discretion.
1652
1653 The tokenBindingId member contains the base64url encoding of the Token
1654 Binding ID that this client uses for the Token Binding protocol when
1655 communicating with the Relying Party. This can be omitted if no Token
1656 Binding has been negotiated between the client and the Relying Party.
1657
1658 The optional clientExtensions and authenticatorExtensions members
1659 contain additional parameters generated by processing the extensions
1660 passed in by the Relying Party. WebAuthn extensions are detailed in
1661 Section 8 WebAuthn Extensions.
1662
1663 This structure is used by the client to compute the following
1664 quantities:
1665
1666 JSON-serialized client data
1667     This is the UTF-8 encoding of the result of calling the initial
1668     value of JSON.stringify on a CollectedClientData dictionary.
1669
1670 Hash of the serialized client data
1671     This is the hash (computed using hashAlgorithm) of the
1672     JSON-serialized client data, as constructed by the client.
1673
1674 4.7.2. Credential Type enumeration (enum PublicKeyCredentialType)
1675
1676 enum PublicKeyCredentialType {
1677     "public-key"
1678 };
1679
1680 This enumeration defines the valid credential types. It is an extension
1681 point; values may be added to it in the future, as more credential
1682 types are defined. The values of this enumeration are used for
1683 versioning the Authentication Assertion and attestation structures
1684 according to the type of the authenticator.
1685
1686 Currently one credential type is defined, namely "public-key".
1687
1688 4.7.3. Credential Descriptor (dictionary PublicKeyCredentialDescriptor)
1689
1690 dictionary PublicKeyCredentialDescriptor {
1691     required PublicKeyCredentialType type;
1692     required BufferSource id;
1693     sequence<AuthenticatorTransport> transports;
1694 };
1695
1696 This dictionary contains the attributes that are specified by a caller
1697 when referring to a credential as an input parameter to the create() or
1698 get() methods. It mirrors the fields of the PublicKeyCredential object
1699 returned by the latter methods.
1700
1701 The type member contains the type of the credential the caller is
1702 referring to.
1703

```

```

1709 required DOMString      origin;
1710 required DOMString      hashAlgorithm;
1711 DOMString                tokenBindingId;
1712 AuthenticationExtensions clientExtensions;
1713 AuthenticationExtensions authenticatorExtensions;
1714 };
1715
1716 The challenge member contains the base64url encoding of the challenge
1717 provided by the RP.
1718
1719 The origin member contains the fully qualified origin of the requester,
1720 as provided to the authenticator by the client, in the syntax defined
1721 by [RFC6454].
1722
1723 The hashAlgorithm member is a recognized algorithm name that supports
1724 the "digest" operation, which specifies the algorithm used to compute
1725 the hash of the serialized client data. This algorithm is chosen by the
1726 client at its sole discretion.
1727
1728 The tokenBindingId member contains the base64url encoding of the Token
1729 Binding ID that this client uses for the Token Binding protocol when
1730 communicating with the Relying Party. This can be omitted if no Token
1731 Binding has been negotiated between the client and the Relying Party.
1732
1733 The optional clientExtensions and authenticatorExtensions members
1734 contain additional parameters generated by processing the extensions
1735 passed in by the Relying Party. WebAuthn extensions are detailed in
1736 Section 9 WebAuthn Extensions.
1737
1738 This structure is used by the client to compute the following
1739 quantities:
1740
1741 JSON-serialized client data
1742     This is the UTF-8 encoding of the result of calling the initial
1743     value of JSON.stringify on a CollectedClientData dictionary.
1744
1745 Hash of the serialized client data
1746     This is the hash (computed using hashAlgorithm) of the
1747     JSON-serialized client data, as constructed by the client.
1748
1749 5.7.2. Credential Type enumeration (enum PublicKeyCredentialType)
1750
1751 enum PublicKeyCredentialType {
1752     "public-key"
1753 };
1754
1755 This enumeration defines the valid credential types. It is an extension
1756 point; values may be added to it in the future, as more credential
1757 types are defined. The values of this enumeration are used for
1758 versioning the Authentication Assertion and attestation structures
1759 according to the type of the authenticator.
1760
1761 Currently one credential type is defined, namely "public-key".
1762
1763 5.7.3. Credential Descriptor (dictionary PublicKeyCredentialDescriptor)
1764
1765 dictionary PublicKeyCredentialDescriptor {
1766     required PublicKeyCredentialType type;
1767     required BufferSource id;
1768     sequence<AuthenticatorTransport> transports;
1769 };
1770
1771 This dictionary contains the attributes that are specified by a caller
1772 when referring to a credential as an input parameter to the create() or
1773 get() methods. It mirrors the fields of the PublicKeyCredential object
1774 returned by the latter methods.
1775
1776 The type member contains the type of the credential the caller is
1777 referring to.

```



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 AuthenticatorTransport {
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  
 1771 a TPM) could be considered more trustworthy than the rest of the  
 1772 authenticator.  
 1773

1778 The id member contains the identifier of the credential that the caller  
 1779 is referring to.  
 1780

1781 **5.7.4. Authenticator Transport enumeration (enum AuthenticatorTransport)**  
 1782

```
1783 enum AuthenticatorTransport {
1784     "usb",
1785     "nfc",
1786     "ble"
1787 };
1788
```

1789 Authenticators may communicate with Clients using a variety of  
 1790 transports. This enumeration defines a hint as to how Clients might  
 1791 communicate with a particular Authenticator in order to obtain an  
 1792 assertion for a specific credential. Note that these hints represent  
 1793 the Relying Party's best belief as to how an Authenticator may be  
 1794 reached. A Relying Party may obtain a list of transports hints from  
 1795 some attestation statement formats or via some out-of-band mechanism;  
 1796 it is outside the scope of this specification to define that mechanism.  
 1797 \* usb - the respective Authenticator may be contacted over USB.  
 1798 \* nfc - the respective Authenticator may be contacted over Near Field  
 1799 Communication (NFC).  
 1800 \* ble - the respective Authenticator may be contacted over Bluetooth  
 1801 Smart (Bluetooth Low Energy / BLE).  
 1802

1803 **5.7.5. Cryptographic Algorithm Identifier (typedef COSEAlgorithmIdentifier)**  
 1804

```
1805 typedef long COSEAlgorithmIdentifier;
1806
```

1807 A COSEAlgorithmIdentifier's value is a number identifying a  
 1808 cryptographic algorithm. The algorithm identifiers SHOULD be values  
 1809 registered in the IANA COSE Algorithms registry [IANA-COSE-ALGS-REG],  
 1810 for instance, -7 for "ES256" and -257 for "RS256".  
 1811

1812 **6. WebAuthn Authenticator model**  
 1813

1814 The API defined in this specification implies a specific abstract  
 1815 functional model for an authenticator. This section describes the  
 1816 authenticator model.  
 1817

1818 Client platforms may implement and expose this abstract model in any  
 1819 way desired. However, the behavior of the client's Web Authentication  
 1820 API implementation, when operating on the authenticators supported by  
 1821 that platform, MUST be indistinguishable from the behavior specified in  
 1822 [5](#) Web Authentication API.  
 1823

1824 For authenticators, this model defines the logical operations that they  
 1825 must support, and the data formats that they expose to the client and  
 1826 the Relying Party. However, it does not define the details of how  
 1827 authenticators communicate with the client platform, unless they are  
 1828 required for interoperability with Relying Parties. For instance, this  
 1829 abstract model does not define protocols for connecting authenticators  
 1830 to clients over transports such as USB or NFC. Similarly, this abstract  
 1831 model does not define specific error codes or methods of returning  
 1832 them; however, it does define error behavior in terms of the needs of  
 1833 the client. Therefore, specific error codes are mentioned as a means of  
 1834 showing which error conditions must be distinguishable (or not) from  
 1835 each other in order to enable a compliant and secure client  
 1836 implementation.  
 1837

1838 In this abstract model, the authenticator provides key management and  
 1839 cryptographic signatures. It may be embedded in the WebAuthn client, or  
 1840 housed in a separate device entirely. The authenticator may itself  
 1841 contain a cryptographic module which operates at a higher security  
 1842 level than the rest of the authenticator. This is particularly  
 1843 important for authenticators that are embedded in the WebAuthn client,  
 1844 as in those cases this cryptographic module (which may, for example, be  
 1845 a TPM) could be considered more trustworthy than the rest of the  
 1846 authenticator.  
 1847

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<sup>-128</sup> 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 2. An assertion signature is produced when the  
1829 authenticatorGetAssertion method is invoked. It represents an  
1830 assertion by the authenticator that the user has consented to a  
1831 specific transaction, such as logging in, or completing a purchase.  
1832 Thus, an assertion signature asserts that the authenticator  
1833 possessing a particular credential private key has established, to  
1834 the best of its ability, that the user requesting this transaction  
1835 is the same user who consented to creating that particular public  
1836 key credential. It also asserts additional information, termed  
1837 client data, that may be useful to the caller, such as the means by  
1838 which user consent was provided, and the prompt shown to the user  
1839 by the authenticator. The assertion signature format is illustrated  
1840 in Figure 2, below.

1841 The formats of these signatures, as well as the procedures for  
1842 generating them, are specified below.  
1843

1848 Each authenticator stores some number of public key credentials. Each  
1849 public key credential has an identifier which is unique (or extremely  
1850 unlikely to be duplicated) among all public key credentials. Each  
1851 credential is also associated with a Relying Party, whose identity is  
1852 represented by a Relying Party Identifier (RP ID).  
1853

1854 Each authenticator has an AAGUID, which is a 128-bit identifier that  
1855 indicates the type (e.g. make and model) of the authenticator. The  
1856 AAGUID MUST be chosen by the manufacturer to be identical across all  
1857 substantially identical authenticators made by that manufacturer, and  
1858 different (with probability 1-2<sup>-128</sup> or greater) from the AAGUIDs of  
1859 all other types of authenticators. The RP MAY use the AAGUID to infer  
1860 certain properties of the authenticator, such as certification level  
1861 and strength of key protection, using information from other sources.  
1862

1863 The primary function of the authenticator is to provide WebAuthn  
1864 signatures, which are bound to various contextual data. These data are  
1865 observed, and added at different levels of the stack as a signature  
1866 request passes from the server to the authenticator. In verifying a  
1867 signature, the server checks these bindings against expected values.  
1868 These contextual bindings are divided in two: Those added by the RP or  
1869 the client, referred to as client data; and those added by the  
1870 authenticator, referred to as the authenticator data. The authenticator  
1871 signs over the client data, but is otherwise not interested in its  
1872 contents. To save bandwidth and processing requirements on the  
1873 authenticator, the client hashes the client data and sends only the  
1874 result to the authenticator. The authenticator signs over the  
1875 combination of the hash of the serialized client data, and its own  
1876 authenticator data.  
1877

- 1878 The goals of this design can be summarized as follows.  
1879 \* The scheme for generating signatures should accommodate cases where  
1880 the link between the client platform and authenticator is very  
1881 limited, in bandwidth and/or latency. Examples include Bluetooth  
1882 Low Energy and Near-Field Communication.  
1883 \* The data processed by the authenticator should be small and easy to  
1884 interpret in low-level code. In particular, authenticators should  
1885 not have to parse high-level encodings such as JSON.  
1886 \* Both the client platform and the authenticator should have the  
1887 flexibility to add contextual bindings as needed.  
1888 \* The design aims to reuse as much as possible of existing encoding  
1889 formats in order to aid adoption and implementation.  
1890

1891 Authenticators produce cryptographic signatures for two distinct  
1892 purposes:

- 1893 1. An attestation signature is produced when a new public key  
1894 credential is created via an authenticatorMakeCredential operation.  
1895 An attestation signature provides cryptographic proof of certain  
1896 properties of the the authenticator and the credential. For  
1897 instance, an attestation signature asserts the authenticator type  
1898 (as denoted by its AAGUID) and the credential public key. The  
1899 attestation signature is signed by an attestation private key,  
1900 which is chosen depending on the type of attestation desired. For  
1901 more details on attestation, see 6.3 Attestation.
- 1902 2. An assertion signature is produced when the  
1903 authenticatorGetAssertion method is invoked. It represents an  
1904 assertion by the authenticator that the user has consented to a  
1905 specific transaction, such as logging in, or completing a purchase.  
1906 Thus, an assertion signature asserts that the authenticator  
1907 possessing a particular credential private key has established, to  
1908 the best of its ability, that the user requesting this transaction  
1909 is the same user who consented to creating that particular public  
1910 key credential. It also asserts additional information, termed  
1911 client data, that may be useful to the caller, such as the means by  
1912 which user consent was provided, and the prompt shown to the user  
1913 by the authenticator. The assertion signature format is illustrated  
1914 in Figure 2, below.

1915 The formats of these signatures, as well as the procedures for  
1916 generating them, are specified below.  
1917

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

1918  
1919 **6.1. Authenticator data**  
1920  
1921 The authenticator data structure encodes contextual bindings made by  
1922 the authenticator. These bindings are controlled by the authenticator  
1923 itself, and derive their trust from the Relying Party's assessment of  
1924 the security properties of the authenticator. In one extreme case, the  
1925 authenticator may be embedded in the client, and its bindings may be no  
1926 more trustworthy than the client data. At the other extreme, the  
1927 authenticator may be a discrete entity with high-security hardware and  
1928 software, connected to the client over a secure channel. In both cases,  
1929 the Relying Party receives the authenticator data in the same format,  
1930 and uses its knowledge of the authenticator to make trust decisions.  
1931  
1932 The authenticator data has a compact but extensible encoding. This is  
1933 desired since authenticators can be devices with limited capabilities  
1934 and low power requirements, with much simpler software stacks than the  
1935 client platform components.  
1936  
1937 The authenticator data structure is a byte array of 37 bytes or more,  
1938 as follows.  
1939  
1940 Length (in bytes) Description  
1941 32 SHA-256 hash of the RP ID associated with the credential.  
1942 1 Flags (bit 0 is the least significant bit):  
1943 \* Bit 0: User Present (UP) result.  
1944 + 1 means the user is present.  
1945 + 0 means the user is not present.  
1946 \* Bit 1: Reserved for future use (RFU1).  
1947 \* Bit 2: User Verified (UV) result.  
1948 + 1 means the user is verified.  
1949 + 0 means the user is not verified.  
1950 \* Bits 3-5: Reserved for future use (RFU2).  
1951 \* Bit 6: Attestation data included (AT).  
1952 + Indicates whether the authenticator added attestation data.  
1953 \* Bit 7: Extension data included (ED).  
1954 + Indicates if the authenticator data has extensions.  
1955  
1956 4 Signature counter (signCount), 32-bit unsigned big-endian integer.  
1957 variable (if present) attestation data (if present). See 6.3.1  
1958 Attestation data for details. Its length depends on the length of the  
1959 credential public key and credential ID being attested.  
1960 variable (if present) Extension-defined authenticator data. This is a  
1961 CBOR [RFC7049] map with extension identifiers as keys, and  
1962 authenticator extension outputs as values. See 9 WebAuthn Extensions  
1963 for details.  
1964  
1965 The RP ID is originally received from the client when the credential is  
1966 created, and again when an assertion is generated. However, it differs  
1967 from other client data in some important ways. First, unlike the client  
1968 data, the RP ID of a credential does not change between operations but  
1969 instead remains the same for the lifetime of that credential. Secondly,  
1970 it is validated by the authenticator during the  
1971 authenticatorGetAssertion operation, by verifying that the RP ID  
1972 associated with the requested credential exactly matches the RP ID  
1973 supplied by the client, and that the RP ID is a registrable domain  
1974 suffix of or is equal to the effective domain of the RP's origin's  
1975 effective domain.  
1976  
1977 The UP flag SHALL be set if and only if the authenticator detected a  
1978 user through an authenticator specific gesture. The RFU bits SHALL be  
1979 set to zero.  
1980  
1981 For attestation signatures, the authenticator MUST set the AT flag and  
1982 include the attestation data. For authentication signatures, the AT  
1983 flag MUST NOT be set and the attestation data MUST NOT be included.  
1984  
1985 If the authenticator does not include any extension data, it MUST set  
1986 the ED flag to zero, and to one if extension data is included.  
1987



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

- 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

1987 The figure below shows a visual representation of the authenticator  
1988 data structure.  
1989 [fido-signature-formats-figure1.svg] Authenticator data layout.  
1990  
1991 Note that the authenticator data describes its own length: If the AT  
1992 and ED flags are not set, it is always 37 bytes long. The attestation  
1993 data (which is only present if the AT flag is set) describes its own  
1994 length. If the ED flag is set, then the total length is 37 bytes plus  
1995 the length of the attestation data, plus the length of the CBOR map  
1996 that follows.  
1997

## 1998 6.1.1. Signature Counter Considerations

1999 Authenticators MUST implement a signature counter feature. The  
2000 signature counter is incremented for each successful  
2001 authenticatorGetAssertion operation by some positive value, and its  
2002 value is returned to the Relying Party within the authenticator data.  
2003 The signature counter's purpose is to aid Relying Parties in detecting  
2004 cloned authenticators. Clone detection is more important for  
2005 authenticators with limited protection measures.  
2006

2007 An Relying Party stores the signature counter of the most recent  
2008 authenticatorGetAssertion operation. Upon a new  
2009 authenticatorGetAssertion operation, the Relying Party compares the  
2010 stored signature counter value with the new signature counter value  
2011 returned in the assertion's authenticator data. If this new signature  
2012 counter value is less than or equal to the stored value, a cloned  
2013 authenticator may exist, or the authenticator may be malfunctioning.  
2014

2015 Detecting a signature counter mismatch does not indicate whether the  
2016 current operation was performed by a cloned authenticator or the  
2017 original authenticator. Relying Parties should address this situation  
2018 appropriately relative to their individual situations, i.e., their risk  
2019 tolerance.  
2020

- 2021 Authenticators:
- 2022 \* should implement per-RP ID signature counters. This prevents the  
2023 signature counter value from being shared between Relying Parties  
2024 and being possibly employed as a correlation handle for the user.  
2025 Authenticators may implement a global signature counter, i.e., on a  
2026 per-authenticator basis, but this is less privacy-friendly for  
2027 users.
  - 2028 \* should ensure that the signature counter value does not  
2029 accidentally decrease (e.g., due to hardware failures).  
2030

## 2031 6.2. Authenticator operations

2032 A client must connect to an authenticator in order to invoke any of the  
2033 operations of that authenticator. This connection defines an  
2034 authenticator session. An authenticator must maintain isolation between  
2035 sessions. It may do this by only allowing one session to exist at any  
2036 particular time, or by providing more complicated session management.  
2037

2038 The following operations can be invoked by the client in an  
2039 authenticator session.  
2040

### 2041 6.2.1. The authenticatorMakeCredential operation

2042 This operation must be invoked in an authenticator session which has no  
2043 other operations in progress. It takes the following input parameters:  
2044

- 2045 **rpId**  
2046 The caller's RP ID, as determined by the user agent and the  
2047 client.
- 2048 **hash**  
2049 The hash of the serialized client data, provided by the client.
- 2050 **rpEntity**  
2051 The Relying Party's PublicKeyCredentialRpEntity.  
2052

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  
 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

2057 userEntity

2058 The user account's PublicKeyCredentialUserEntity, containing the  
 2059 user handle given by the Relying Party.

2062 credTypesAndPubKeyAlgs

2063 A sequence of pairs of PublicKeyCredentialType and public key  
 2064 algorithms (COSEAlgorithmIdentifier) requested by the Relying  
 2065 Party. This sequence is ordered from most preferred to least  
 2066 preferred. The platform makes a best-effort to create the most  
 2067 preferred credential that it can.

2070 excludeCredentialDescriptorList

2071 An optional list of PublicKeyCredentialDescriptor objects  
 2072 provided by the Relying Party with the intention that, if any of  
 2073 these are known to the authenticator, it should not create a new  
 2074 credential. excludeCredentialDescriptorList contains a list of  
 2075 known credentials.

2076 requireResidentKey

2077 options.authenticatorSelection.requireResidentKey.

2079 requireUserVerification

2080 options.authenticatorSelection.requireUserVerification

2082 extensions

2083 A map from extension identifiers to their authenticator  
 2084 extension inputs, created by the client based on the extensions  
 2085 requested by the Relying Party, if any.

2086  
 2087 When this operation is invoked, the authenticator must perform the  
 2088 following procedure:

- 2089 1. Check if all the supplied parameters are syntactically well-formed  
 2090 and of the correct length. If not, return an error code equivalent  
 2091 to "UnknownError" and terminate the operation.
- 2092 2. Check if at least one of the specified combinations of  
 2093 PublicKeyCredentialType and cryptographic parameters in  
 2094 credTypesAndPubKeyAlgs is supported. If not, return an error code  
 2095 equivalent to "NotSupportedError" and terminate the operation.
- 2096 3. Check if a credential matching an item of  
 2097 excludeCredentialDescriptorList is present on this authenticator.  
 2098 If so, return an error code equivalent to "NotAllowedError" and  
 2099 terminate the operation.
- 2100 4. If requireResidentKey is true and the authenticator cannot store a  
 2101 Client-side-resident Credential Private Key, return an error code  
 2102 equivalent to "ConstraintError" and terminate the operation.
- 2103 5. If requireUserVerification is true and the authenticator cannot  
 2104 perform user verification, return an error code equivalent to  
 2105 "ConstraintError" and terminate the operation.
- 2106 6. Prompt the user for consent to create a new credential. The prompt  
 2107 for obtaining this consent is shown by the authenticator if it has  
 2108 its own output capability, or by the user agent otherwise. If the  
 2109 user denies consent, return an error code equivalent to  
 2110 "NotAllowedError" and terminate the operation. The Authenticator  
 2111 and user agent MAY skip this prompt if the Authenticator is a  
 2112 platform authenticator and excludeCredentialDescriptorList is  
 2113 empty.
- 2114 7. Once user consent has been obtained, generate a new credential  
 2115 object:
  - 2116 1. Let (publicKey.privateKey) be a new set of cryptographic keys  
 2117 using the combination of PublicKeyCredentialType and  
 2118 cryptographic parameters represented by the first item in  
 2119 credTypesAndPubKeyAlgs that is supported by this  
 2120 authenticator.
  - 2121 2. Let credentialId be a new identifier for this credential that  
 2122 is globally unique with high probability across all  
 2123 credentials with the same type across all authenticators.
  - 2124 3. Let userHandle be userEntity.id.
  - 2125 4. Associate the credentialId and privateKey with rpId and  
 2126 userHandle.

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 On successful completion of this operation, the authenticator returns

2004 the attestation object to the client.

2005

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 When this method is invoked, the authenticator must perform the

2018 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

- 2127 5. Delete any older credentials with the same rpId and userHandle
- 2128 that are stored locally by the authenticator.
- 2129 8. If any error occurred while creating the new credential object,
- 2130 return an error code equivalent to "UnknownError" and terminate the
- 2131 operation.
- 2132 9. Let processedExtensions be the result of authenticator extension
- 2133 processing for each supported extension identifier/input pair in
- 2134 extensions.
- 2135 10. If the authenticator supports:
- 2136
- 2137 a per-RP ID signature counter
- 2138 allocate the counter, associate it with the RP ID, and
- 2139 initialize the counter value as zero.
- 2140
- 2141 a global signature counter
- 2142 Use the global signature counter's actual value when
- 2143 generating authenticator data.
- 2144
- 2145 a per credential signature counter
- 2146 allocate the counter, associate it with the new
- 2147 credential, and initialize the counter value as zero.
- 2148
- 2149 11. Let attestationData be the attestation data byte array including
- 2150 the credentialId and publicKey.
- 2151 12. Let authenticatorData be the byte array specified in 6.1
- 2152 Authenticator data including attestationData and any
- 2153 processedExtensions.
- 2154 13. Return the attestation object for the new credential created by the
- 2155 procedure specified in 6.3.4 Generating an Attestation Object
- 2156 using an authenticator-chosen attestation statement format,
- 2157 authenticatorData, and hash. For more details on attestation, see
- 2158 6.3 Attestation.

2159 On successful completion of this operation, the authenticator returns

2160 the attestation object to the client.

2161

2162

2163 **6.2.2. The authenticatorGetAssertion operation**

2164

2165 This operation must be invoked in an authenticator session which has no

2166 other operations in progress. It takes the following input parameters:

- 2167 \* The caller's RP ID, as determined by the user agent and the client.
- 2168 \* The hash of the serialized client data, provided by the client.
- 2169 \* A list of credentials acceptable to the Relying Party (possibly
- 2170 filtered by the client), if any.
- 2171 \* Extension data created by the client based on the extensions
- 2172 requested by the Relying Party, if any.

2173 When this method is invoked, the authenticator must perform the

2174 following procedure:

- 2175
- 2176 1. Check if all the supplied parameters are syntactically well-formed
- 2177 and of the correct length. If not, return an error code equivalent
- 2178 to "UnknownError" and terminate the operation.
- 2179 2. If a list of credentials was supplied by the client, filter it by
- 2180 removing those credentials that are not present on this
- 2181 authenticator. If no list was supplied, create a list with all
- 2182 credentials stored for the caller's RP ID (as determined by an
- 2183 exact match of the RP ID).
- 2184 3. If the previous step resulted in an empty list, return an error
- 2185 code equivalent to "NotAllowedError" and terminate the operation.
- 2186 4. Prompt the user to select a credential from among the above list.
- 2187 Obtain user consent for using this credential. The prompt for
- 2188 obtaining this consent may be shown by the authenticator if it has
- 2189 its own output capability, or by the user agent otherwise.
- 2190 5. Process all the supported extensions requested by the client.
- 2191 6. Increment the RP ID-associated signature counter or the global
- 2192 signature counter value, depending on which approach is implemented
- 2193 by the authenticator by some positive value.
- 2194 7. Generate the authenticator data as specified in 6.1 Authenticator
- 2195 data, though without attestation data.
- 2196 8. Concatenate this authenticator data with the hash of the serialized



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.

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

2050 On successful completion, the authenticator returns to the user agent:  
2051 \* The identifier of the credential (credential ID) used to generate  
2052 the assertion signature.  
2053 \* The authenticator data used to generate the assertion signature.  
2054 \* The assertion signature.

2055 If the authenticator cannot find any credential corresponding to the  
2056 specified Relying Party that matches the specified criteria, it  
2057 terminates the operation and returns an error.

2060 If the user refuses consent, the authenticator returns an appropriate  
2061 error status to the client.

### 2063 5.2.3. The authenticatorCancel operation

2064 This operation takes no input parameters and returns no result.

2067 When this operation is invoked by the client in an authenticator  
2068 session, it has the effect of terminating any  
2069 authenticatorMakeCredential or authenticatorGetAssertion operation  
2070 currently in progress in that authenticator session. The authenticator  
2071 stops prompting for, or accepting, any user input related to  
2072 authorizing the canceled operation. The client ignores any further  
2073 responses from the authenticator for the canceled operation.

2075 This operation is ignored if it is invoked in an authenticator session  
2076 which does not have an authenticatorMakeCredential or  
2077 authenticatorGetAssertion operation currently in progress.

### 2079 5.3. Attestation

2081 Authenticators must also provide some form of attestation. The basic  
2082 requirement is that the authenticator can produce, for each credential  
2083 public key, an attestation statement verifiable by the Relying Party.  
2084 Typically, this attestation statement contains a signature by an  
2085 attestation private key over the attested credential public key and a  
2086 challenge, as well as a certificate or similar data providing  
2087 provenance information for the attestation public key, enabling the  
2088 Relying Party to make a trust decision. However, if an attestation key  
2089 pair is not available, then the authenticator MUST perform self  
2090 attestation of the credential public key with the corresponding  
2091 credential private key. All this information is returned by  
2092 authenticators any time a new public key credential is generated, in  
2093 the overall form of an attestation object. The relationship of the  
2094 attestation object with authenticator data (containing attestation  
2095 data) and the attestation statement is illustrated in figure 3, below.  
2096 **Attestation Object Layout diagram** Attestation object layout  
2097 illustrating the included authenticator data (containing attestation  
2098 data) and the attestation statement.

2100 This figure illustrates only the packed attestation statement format.  
2101 Several additional attestation statement formats are defined in **7**  
2102 Defined Attestation Statement Formats.

2104 An important component of the attestation object is the attestation  
2105 statement. This is a specific type of signed data object, containing

2197 client data to generate an assertion signature using the private  
2198 key of the selected credential as shown in Figure 2, below. A  
2199 simple, unlimited concatenation is safe to use here because the  
2200 authenticator data describes its own length. The hash of the  
2201 serialized client data (which potentially has a variable length) is  
2202 always the last element.  
2203 9. If any error occurred while generating the assertion signature,  
2204 return an error code equivalent to "UnknownError" and terminate the  
2205 operation.

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

2209 On successful completion, the authenticator returns to the user agent:  
2210 \* The identifier of the credential (credential ID) used to generate  
2211 the assertion signature.  
2212 \* The authenticator data used to generate the assertion signature.  
2213 \* The assertion signature.  
2214 \* The user handle associated with the credential used to generate the  
2215 assertion signature.

2217 If the authenticator cannot find any credential corresponding to the  
2218 specified Relying Party that matches the specified criteria, it  
2219 terminates the operation and returns an error.

2221 If the user refuses consent, the authenticator returns an appropriate  
2222 error status to the client.

### 2224 6.2.3. The authenticatorCancel operation

2225 This operation takes no input parameters and returns no result.

2228 When this operation is invoked by the client in an authenticator  
2229 session, it has the effect of terminating any  
2230 authenticatorMakeCredential or authenticatorGetAssertion operation  
2231 currently in progress in that authenticator session. The authenticator  
2232 stops prompting for, or accepting, any user input related to  
2233 authorizing the canceled operation. The client ignores any further  
2234 responses from the authenticator for the canceled operation.

2236 This operation is ignored if it is invoked in an authenticator session  
2237 which does not have an authenticatorMakeCredential or  
2238 authenticatorGetAssertion operation currently in progress.

### 2240 6.3. Attestation

2241 Authenticators must also provide some form of attestation. The basic  
2242 requirement is that the authenticator can produce, for each credential  
2243 public key, an attestation statement verifiable by the Relying Party.  
2244 Typically, this attestation statement contains a signature by an  
2245 attestation private key over the attested credential public key and a  
2246 challenge, as well as a certificate or similar data providing  
2247 provenance information for the attestation public key, enabling the  
2248 Relying Party to make a trust decision. However, if an attestation key  
2249 pair is not available, then the authenticator MUST perform self  
2250 attestation of the credential public key with the corresponding  
2251 credential private key. All this information is returned by  
2252 authenticators any time a new public key credential is generated, in  
2253 the overall form of an attestation object. The relationship of the  
2254 attestation object with authenticator data (containing attestation  
2255 data) and the attestation statement is illustrated in figure 3, below.  
2256 **[fido-attestation-structures.svg]** Attestation object layout  
2257 illustrating the included authenticator data (containing attestation  
2258 data) and the attestation statement.

2260 This figure illustrates only the packed attestation statement format.  
2261 Several additional attestation statement formats are defined in **8**  
2262 Defined Attestation Statement Formats.

2264 An important component of the attestation object is the attestation  
2265 statement. This is a specific type of signed data object, containing

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

1. The attestation statement format is the manner in which the signature is represented and the various contextual bindings are incorporated into the attestation statement by the authenticator. In other words, this defines the syntax of the statement. Various existing devices and platforms (such as TPMs and the Android OS) have previously defined attestation statement formats. This specification supports a variety of such formats in an extensible way, as defined in 5.3.2 Attestation Statement Formats.
2. The attestation type defines the semantics of attestation statements and their underlying trust models. Specifically, it defines how a Relying Party establishes trust in a particular attestation statement, after verifying that it is cryptographically valid. This specification supports a number of attestation types, as described in 5.3.3 Attestation Types.

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

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

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

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

### 5.3.1. Attestation data

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

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

### 5.3.2. Attestation Statement Formats

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

- \* Attestation statement format identifier:
- \* Supported attestation types:
- \* Syntax: The syntax of an attestation statement produced in this

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

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

In general, there is no simple mapping between attestation statement formats and attestation types. For example, the "packed" attestation statement format defined in 8.2 Packed Attestation Statement Format can be used in conjunction with all attestation types, while other formats and types have more limited applicability.

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

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

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

### 6.3.1. 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.

### 6.3.2. Attestation Statement Formats

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

- \* Attestation statement format identifier:
- \* Supported attestation types:
- \* Syntax: The syntax of an attestation statement produced in this

2176 format, defined using [CDDL] for the extension point \$attStmtFormat  
 2177 defined in 5.3.4 Generating an Attestation Object.  
 2178 \* Signing procedure: The signing procedure for computing an  
 2179 attestation statement in this format given the public key  
 2180 credential to be attested, the authenticator data structure  
 2181 containing the authenticator data for the attestation, and the hash  
 2182 of the serialized client data.  
 2183 \* Verification procedures: The procedure for verifying an attestation  
 2184 statement, which takes as inputs the authenticator data structure  
 2185 containing the authenticator data claimed to have been used for the  
 2186 attestation and the hash of the serialized client data, and returns  
 2187 either:  
 2188 + An error indicating that the attestation is invalid, or  
 2189 + The attestation type, and the trust path of the attestation.  
 2190 This trust path is either empty (in case of self attestation),  
 2191 an identifier of a ECDAA-Issuer public key (in the case of  
 2192 ECDAA), or a set of X.509 certificates.

2194 The initial list of specified attestation statement formats is in 7  
 2195 Defined Attestation Statement Formats.

2197 **5.3.3. Attestation Types**

2198 WebAuthn supports multiple attestation types:

2201 **Basic Attestation**

2202 In the case of basic attestation [UAFProtocol], the  
 2203 authenticator's attestation key pair is specific to an  
 2204 authenticator model. Thus, authenticators of the same model  
 2205 often share the same attestation key pair. See 5.3.5.1 Privacy  
 2206 for futher information.

2208 **Self Attestation**

2209 In the case of self attestation, also known as surrogate basic  
 2210 attestation [UAFProtocol], the Authenticator does not have any  
 2211 specific attestation key. Instead it uses the authentication key  
 2212 itself to create the attestation signature. Authenticators  
 2213 without meaningful protection measures for an attestation  
 2214 private key typically use this attestation type.

2216 **Privacy CA**

2217 In this case, the Authenticator owns an authenticator-specific  
 2218 (endorsement) key. This key is used to securely communicate with  
 2219 a trusted third party, the Privacy CA. The Authenticator can  
 2220 generate multiple attestation key pairs and asks the Privacy CA  
 2221 to issue an attestation certificate for it. Using this approach,  
 2222 the Authenticator can limit the exposure of the endorsement key  
 2223 (which is a global correlation handle) to Privacy CA(s).  
 2224 Attestation keys can be requested for each public key credential  
 2225 individually.

2226 Note: This concept typically leads to multiple attestation  
 2227 certificates. The attestation certificate requested most  
 2228 recently is called "active".

2231 **Elliptic Curve based Direct Anonymous Attestation (ECDAA)**

2232 In this case, the Authenticator receives direct anonymous  
 2233 attestation (DAA) credentials from a single DAA-Issuer. These  
 2234 DAA credentials are used along with blinding to sign the  
 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]).

2242 **5.3.4. Generating an Attestation Object**

2243 This section specifies the algorithm for generating an attestation  
 2244 object (see: Figure 3) for any attestation statement format.  
 2245

2335 format, defined using [CDDL] for the extension point \$attStmtFormat  
 2336 defined in 6.3.4 Generating an Attestation Object.  
 2337 \* Signing procedure: The signing procedure for computing an  
 2338 attestation statement in this format given the public key  
 2339 credential to be attested, the authenticator data structure  
 2340 containing the authenticator data for the attestation, and the hash  
 2341 of the serialized client data.  
 2342 \* Verification procedures: The procedure for verifying an attestation  
 2343 statement, which takes as inputs the authenticator data structure  
 2344 containing the authenticator data claimed to have been used for the  
 2345 attestation and the hash of the serialized client data, and returns  
 2346 either:  
 2347 + An error indicating that the attestation is invalid, or  
 2348 + The attestation type, and the trust path of the attestation.  
 2349 This trust path is either empty (in case of self attestation),  
 2350 an identifier of a ECDAA-Issuer public key (in the case of  
 2351 ECDAA), or a set of X.509 certificates.

2353 The initial list of specified attestation statement formats is in 8  
 2354 Defined Attestation Statement Formats.

2356 **6.3.3. Attestation Types**

2357 WebAuthn supports multiple attestation types:

2360 **Basic Attestation**

2361 In the case of basic attestation [UAFProtocol], the  
 2362 authenticator's attestation key pair is specific to an  
 2363 authenticator model. Thus, authenticators of the same model  
 2364 often share the same attestation key pair. See 6.3.5.1 Privacy  
 2365 for futher information.

2367 **Self Attestation**

2368 In the case of self attestation, also known as surrogate basic  
 2369 attestation [UAFProtocol], the Authenticator does not have any  
 2370 specific attestation key. Instead it uses the authentication key  
 2371 itself to create the attestation signature. Authenticators  
 2372 without meaningful protection measures for an attestation  
 2373 private key typically use this attestation type.

2375 **Privacy CA**

2376 In this case, the Authenticator owns an authenticator-specific  
 2377 (endorsement) key. This key is used to securely communicate with  
 2378 a trusted third party, the Privacy CA. The Authenticator can  
 2379 generate multiple attestation key pairs and asks the Privacy CA  
 2380 to issue an attestation certificate for it. Using this approach,  
 2381 the Authenticator can limit the exposure of the endorsement key  
 2382 (which is a global correlation handle) to Privacy CA(s).  
 2383 Attestation keys can be requested for each public key credential  
 2384 individually.

2385 Note: This concept typically leads to multiple attestation  
 2386 certificates. The attestation certificate requested most  
 2387 recently is called "active".

2390 **Elliptic Curve based Direct Anonymous Attestation (ECDAA)**

2391 In this case, the Authenticator receives direct anonymous  
 2392 attestation (DAA) credentials from a single DAA-Issuer. These  
 2393 DAA credentials are used along with blinding to sign the  
 2394 attestation data. The concept of blinding avoids the DAA  
 2395 credentials being misused as global correlation handle. WebAuthn  
 2396 supports DAA using elliptic curve cryptography and bilinear  
 2397 pairings, called ECDAA (see [FIDOEcdaaAlgorithm]) in this  
 2398 specification. Consequently we denote the DAA-Issuer as  
 2399 ECDAA-Issuer (see [FIDOEcdaaAlgorithm]).

2402 **6.3.4. Generating an Attestation Object**

2403 To generate an attestation object (see: Figure 3) given:



2246 In order to construct an attestation object for a given public key  
 2247 credential using a particular attestation statement format, the  
 2248 authenticator MUST first generate the authenticator data.  
 2249  
 2250 The authenticator MUST then run the signing procedure for the desired  
 2251 attestation statement format with this authenticator data and the hash  
 2252 of the serialized client data as input, and use this to construct an  
 2253 attestation statement in that attestation statement format.  
 2254  
 2255 Finally, the authenticator MUST construct the attestation object as a  
 2256 CBOR map with the following syntax:  
 2257

```
2258 attObj = {
2259   authData: bytes,
2260   $$attStmtType
2261 }
2262
2263 attStmtTemplate = (
2264   fmt: text,
2265   attStmt: bytes
2266 )
```

2267 ; Every attestation statement format must have the above fields  
 2268 attStmtTemplate .within \$\$attStmtType

2270 The semantics of the fields in the attestation object are as follows:

2271 **fmt**  
 2272 The attestation statement format identifier associated with the  
 2273 attestation statement. Each attestation statement format defines  
 2274 its identifier.

2275 **authData**  
 2276 The authenticator data used to generate the attestation  
 2277 statement.

2278 **attStmt**  
 2279 The attestation statement constructed above. The syntax of this  
 2280 is defined by the attestation statement format used.

### 2281 5.3.5. Security Considerations

#### 2282 5.3.5.1. Privacy

2283 Attestation keys may be used to track users or link various online  
 2284 identities of the same user together. This may be mitigated in several  
 2285 ways, including:

- 2286 \* A WebAuthn authenticator manufacturer may choose to ship all of  
 2287 their devices with the same (or a fixed number of) attestation  
 2288 key(s) (called Basic Attestation). This will anonymize the user at  
 2289 the risk of not being able to revoke a particular attestation key  
 2290 should its WebAuthn Authenticator be compromised.
- 2291 \* A WebAuthn Authenticator may be capable of dynamically generating  
 2292 different attestation keys (and requesting related certificates)  
 2293 per origin (following the Privacy CA approach). For example, a  
 2294 WebAuthn Authenticator can ship with a master attestation key (and  
 2295 certificate), and combined with a cloud operated privacy CA, can  
 2296 dynamically generate per origin attestation keys and attestation  
 2297 certificates.
- 2298 \* A WebAuthn Authenticator can implement Elliptic Curve based direct  
 2299 anonymous attestation (see [FIDOEcdaaAlgorithm]). Using this

2404 **attestationFormat**  
 2405 An attestation statement format.  
 2406

2407 **authData**  
 2408 A byte array containing authenticator data.  
 2409

2410 **hash**  
 2411 The hash of the serialized client data.  
 2412

2413 the authenticator MUST:  
 2414 1. Let attStmt be the result of running attestationFormat's signing  
 2415 procedure given authData and hash.  
 2416 2. Let fmt be attestationFormat's attestation statement format  
 2417 identifier  
 2418 3. Return the attestation object as a CBOR map with the following  
 2419 syntax, filled in with variables initialized by this algorithm:

```
2420 attObj = {
2421   authData: bytes,
2422   $$attStmtType
2423 }
2424
2425 attStmtTemplate = (
2426   fmt: text,
2427   attStmt: { * tstr => any } ; Map is filled in by each
2428 concrete attStmtType
2429 )
```

2430 ; Every attestation statement format must have the above fields  
 2431 attStmtTemplate .within \$\$attStmtType

### 2432 6.3.5. Security Considerations

#### 2433 6.3.5.1. Privacy

2434 Attestation keys may be used to track users or link various online  
 2435 identities of the same user together. This may be mitigated in several  
 2436 ways, including:

- 2437 \* A WebAuthn authenticator manufacturer may choose to ship all of  
 2438 their devices with the same (or a fixed number of) attestation  
 2439 key(s) (called Basic Attestation). This will anonymize the user at  
 2440 the risk of not being able to revoke a particular attestation key  
 2441 should its WebAuthn Authenticator be compromised.
- 2442 \* A WebAuthn Authenticator may be capable of dynamically generating  
 2443 different attestation keys (and requesting related certificates)  
 2444 per origin (following the Privacy CA approach). For example, a  
 2445 WebAuthn Authenticator can ship with a master attestation key (and  
 2446 certificate), and combined with a cloud operated privacy CA, can  
 2447 dynamically generate per origin attestation keys and attestation  
 2448 certificates.
- 2449 \* A WebAuthn Authenticator can implement Elliptic Curve based direct  
 2450 anonymous attestation (see [FIDOEcdaaAlgorithm]). Using this

2307 scheme, the authenticator generates a blinded attestation  
2308 signature. This allows the Relying Party to verify the signature  
2309 using the ECDAA-Issuer public key, but the attestation signature  
2310 does not serve as a global correlation handle.  
2311

2312 **5.3.5.2. Attestation Certificate and Attestation Certificate CA Compromise**  
2313

2314 When an intermediate CA or a root CA used for issuing attestation  
2315 certificates is compromised, WebAuthn authenticator attestation keys  
2316 are still safe although their certificates can no longer be trusted. A  
2317 WebAuthn Authenticator manufacturer that has recorded the public  
2318 attestation keys for their devices can issue new attestation  
2319 certificates for these keys from a new intermediate CA or from a new  
2320 root CA. If the root CA changes, the Relying Parties must update their  
2321 trusted root certificates accordingly.  
2322

2323 A WebAuthn Authenticator attestation certificate must be revoked by the  
2324 issuing CA if its key has been compromised. A WebAuthn Authenticator  
2325 manufacturer may need to ship a firmware update and inject new  
2326 attestation keys and certificates into already manufactured WebAuthn  
2327 Authenticators, if the exposure was due to a firmware flaw. (The  
2328 process by which this happens is out of scope for this specification.)  
2329 If the WebAuthn Authenticator manufacturer does not have this  
2330 capability, then it may not be possible for Relying Parties to trust  
2331 any further attestation statements from the affected WebAuthn  
2332 Authenticators.  
2333

2334 If attestation certificate validation fails due to a revoked  
2335 intermediate attestation CA certificate, and the Relying Party's policy  
2336 requires rejecting the registration/authentication request in these  
2337 situations, then it is recommended that the Relying Party also  
2338 un-registers (or marks with a trust level equivalent to "self  
2339 attestation") public key credentials that were registered after the CA  
2340 compromise date using an attestation certificate chaining up to the  
2341 same intermediate CA. It is thus recommended that Relying Parties  
2342 remember intermediate attestation CA certificates during Authenticator  
2343 registration in order to un-register related public key credentials if  
2344 the registration was performed after revocation of such certificates.  
2345

2346 If an ECDAA attestation key has been compromised, it can be added to  
2347 the RogueList (i.e., the list of revoked authenticators) maintained by  
2348 the related ECDAA-Issuer. The Relying Party should verify whether an  
2349 authenticator belongs to the RogueList when performing ECDAA-Verify  
2350 (see section 3.6 in [FIDOEcdaaAlgorithm]). For example, the FIDO  
2351 Metadata Service [FIDOMetadataService] provides one way to access such  
2352 information.  
2353

2354 **5.3.5.3. Attestation Certificate Hierarchy**  
2355

2356 A 3-tier hierarchy for attestation certificates is recommended (i.e.,  
2357 Attestation Root, Attestation Issuing CA, Attestation Certificate). It  
2358 is also recommended that for each WebAuthn Authenticator device line  
2359 (i.e., model), a separate issuing CA is used to help facilitate  
2360 isolating problems with a specific version of a device.  
2361

2362 If the attestation root certificate is not dedicated to a single  
2363 WebAuthn Authenticator device line (i.e., AAGUID), the AAGUID should be  
2364 specified in the attestation certificate itself, so that it can be  
2365 verified against the authenticator data.  
2366

2367 **6. Relying Party Operations**  
2368

2369 Upon successful execution of create() or get(), the Relying Party's  
2370 script receives a PublicKeyCredential containing an  
2371 AuthenticatorAttestationResponse or AuthenticatorAssertionResponse  
2372 structure, respectively, from the client. It must then deliver the  
2373 contents of this structure to the Relying Party server, using methods  
2374 outside the scope of this specification. This section describes the  
2375 operations that the Relying Party must perform upon receipt of these  
2376 structures.

2456 scheme, the authenticator generates a blinded attestation  
2457 signature. This allows the Relying Party to verify the signature  
2458 using the ECDAA-Issuer public key, but the attestation signature  
2459 does not serve as a global correlation handle.  
2460

2461 **6.3.5.2. Attestation Certificate and Attestation Certificate CA Compromise**  
2462

2463 When an intermediate CA or a root CA used for issuing attestation  
2464 certificates is compromised, WebAuthn authenticator attestation keys  
2465 are still safe although their certificates can no longer be trusted. A  
2466 WebAuthn Authenticator manufacturer that has recorded the public  
2467 attestation keys for their devices can issue new attestation  
2468 certificates for these keys from a new intermediate CA or from a new  
2469 root CA. If the root CA changes, the Relying Parties must update their  
2470 trusted root certificates accordingly.  
2471

2472 A WebAuthn Authenticator attestation certificate must be revoked by the  
2473 issuing CA if its key has been compromised. A WebAuthn Authenticator  
2474 manufacturer may need to ship a firmware update and inject new  
2475 attestation keys and certificates into already manufactured WebAuthn  
2476 Authenticators, if the exposure was due to a firmware flaw. (The  
2477 process by which this happens is out of scope for this specification.)  
2478 If the WebAuthn Authenticator manufacturer does not have this  
2479 capability, then it may not be possible for Relying Parties to trust  
2480 any further attestation statements from the affected WebAuthn  
2481 Authenticators.  
2482

2483 If attestation certificate validation fails due to a revoked  
2484 intermediate attestation CA certificate, and the Relying Party's policy  
2485 requires rejecting the registration/authentication request in these  
2486 situations, then it is recommended that the Relying Party also  
2487 un-registers (or marks with a trust level equivalent to "self  
2488 attestation") public key credentials that were registered after the CA  
2489 compromise date using an attestation certificate chaining up to the  
2490 same intermediate CA. It is thus recommended that Relying Parties  
2491 remember intermediate attestation CA certificates during Authenticator  
2492 registration in order to un-register related public key credentials if  
2493 the registration was performed after revocation of such certificates.  
2494

2495 If an ECDAA attestation key has been compromised, it can be added to  
2496 the RogueList (i.e., the list of revoked authenticators) maintained by  
2497 the related ECDAA-Issuer. The Relying Party should verify whether an  
2498 authenticator belongs to the RogueList when performing ECDAA-Verify  
2499 (see section 3.6 in [FIDOEcdaaAlgorithm]). For example, the FIDO  
2500 Metadata Service [FIDOMetadataService] provides one way to access such  
2501 information.  
2502

2503 **6.3.5.3. Attestation Certificate Hierarchy**  
2504

2505 A 3-tier hierarchy for attestation certificates is recommended (i.e.,  
2506 Attestation Root, Attestation Issuing CA, Attestation Certificate). It  
2507 is also recommended that for each WebAuthn Authenticator device line  
2508 (i.e., model), a separate issuing CA is used to help facilitate  
2509 isolating problems with a specific version of a device.  
2510

2511 If the attestation root certificate is not dedicated to a single  
2512 WebAuthn Authenticator device line (i.e., AAGUID), the AAGUID should be  
2513 specified in the attestation certificate itself, so that it can be  
2514 verified against the authenticator data.  
2515

2516 **7. Relying Party Operations**  
2517

2518 Upon successful execution of create() or get(), the Relying Party's  
2519 script receives a PublicKeyCredential containing an  
2520 AuthenticatorAttestationResponse or AuthenticatorAssertionResponse  
2521 structure, respectively, from the client. It must then deliver the  
2522 contents of this structure to the Relying Party server, using methods  
2523 outside the scope of this specification. This section describes the  
2524 operations that the Relying Party must perform upon receipt of these  
2525 structures.

2377  
2378  
2379  
2380  
2381  
2382  
2383  
2384  
2385  
2386  
2387  
2388  
2389  
2390  
2391  
2392  
2393  
2394  
2395  
2396  
2397  
2398  
2399  
2400  
2401  
2402  
2403  
2404  
2405  
2406  
2407  
2408  
2409  
2410  
2411  
2412  
2413  
2414  
2415  
2416  
2417  
2418  
2419  
2420  
2421  
2422  
2423  
2424  
2425  
2426  
2427  
2428  
2429  
2430  
2431  
2432  
2433  
2434  
2435  
2436  
2437  
2438  
2439  
2440  
2441  
2442  
2443  
2444  
2445  
2446

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.
15. **If verification of the attestation statement failed, the Relying Party MUST fail the registration ceremony.**

2526  
2527  
2528  
2529  
2530  
2531  
2532  
2533  
2534  
2535  
2536  
2537  
2538  
2539  
2540  
2541  
2542  
2543  
2544  
2545  
2546  
2547  
2548  
2549  
2550  
2551  
2552  
2553  
2554  
2555  
2556  
2557  
2558  
2559  
2560  
2561  
2562  
2563  
2564  
2565  
2566  
2567  
2568  
2569  
2570  
2571  
2572  
2573  
2574  
2575  
2576  
2577  
2578  
2579  
2580  
2581  
2582  
2583  
2584  
2585  
2586  
2587  
2588  
2589  
2590  
2591  
2592  
2593

7.1. Registering a new credential

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

1. Perform JSON deserialization on the clientDataJSON field of the AuthenticatorAttestationResponse object to extract the client data C claimed as collected during the credential creation.
2. Verify that the 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 subset of the **extensions requested by the RP and that the authenticatorExtensions in C is also** a subset of the extensions requested by the RP.
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, **and the signature counter contained in authData**, 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 6.3.3 Attestation Types). If doing so, the Relying Party is asserting there is no cryptographic proof that the public key credential has been generated by a particular authenticator model. See [FIDOSecRef] and [UAFProtocol] for a more detailed discussion.



2447 Verification of attestation objects requires that the Relying Party has  
 2448 a trusted method of determining acceptable trust anchors in step 11  
 2449 above. Also, if certificates are being used, the Relying Party must  
 2450 have access to certificate status information for the intermediate CA  
 2451 certificates. The Relying Party must also be able to build the  
 2452 attestation certificate chain if the client did not provide this chain  
 2453 in the attestation information.  
 2454  
 2455 To avoid ambiguity during authentication, the Relying Party SHOULD  
 2456 check that each credential is registered to no more than one user. If  
 2457 registration is requested for a credential that is already registered  
 2458 to a different user, the Relying Party SHOULD fail this ceremony, or it  
 2459 MAY decide to accept the registration, e.g. while deleting the older  
 2460 registration.  
 2461  
 2462 **6.2. Verifying an authentication assertion**  
 2463  
 2464 When verifying a given PublicKeyCredential structure (credential) as  
 2465 part of an authentication ceremony, the Relying Party MUST proceed as  
 2466 follows:  
 2467  
 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  
 2472 2. Let cData, aData and sig denote the value of credential's  
 2473 response's clientDataJSON, authenticatorData, and signature  
 2474 respectively.  
 2475  
 2476 3. Perform JSON deserialization on cData to extract the client data C  
 2477 used for the signature.  
 2478  
 2479 4. Verify that the challenge member of C matches the challenge that  
 2480 was sent to the authenticator in the  
 2481 PublicKeyCredentialRequestOptions passed to the get() call.  
 2482  
 2483 5. Verify that the origin member of C matches the Relying Party's  
 2484 origin.  
 2485  
 2486 6. Verify that the tokenBindingId member of C (if present) matches the  
 2487 Token Binding ID for the TLS connection over which the signature  
 2488 was obtained.  
 2489  
 2490 7. Verify that the clientExtensions member of C is a proper subset of  
 2491 the extensions requested by the Relying Party and that the  
 2492 authenticatorExtensions in C is also a proper subset of the  
 2493 extensions requested by the Relying Party.  
 2494  
 2495 8. Verify that the RP ID hash in aData is the SHA-256 hash of the RP  
 2496 ID expected by the Relying Party.  
 2497  
 2498 9. Let hash be the result of computing a hash over the cData using the  
 2499 algorithm represented by the hashAlgorithm member of C.  
 2500  
 2501 10. Using the credential public key looked up in step 1, verify that  
 2502 sig is a valid signature over the binary concatenation of aData and  
 2503 hash.  
 2504  
 2505 11. If all the above steps are successful, continue with the

2594 Verification of attestation objects requires that the Relying Party has  
 2595 a trusted method of determining acceptable trust anchors in step 11  
 2596 above. Also, if certificates are being used, the Relying Party must  
 2597 have access to certificate status information for the intermediate CA  
 2598 certificates. The Relying Party must also be able to build the  
 2599 attestation certificate chain if the client did not provide this chain  
 2600 in the attestation information.  
 2601  
 2602 To avoid ambiguity during authentication, the Relying Party SHOULD  
 2603 check that each credential is registered to no more than one user. If  
 2604 registration is requested for a credential that is already registered  
 2605 to a different user, the Relying Party SHOULD fail this ceremony, or it  
 2606 MAY decide to accept the registration, e.g. while deleting the older  
 2607 registration.  
 2608  
 2609 **7.2. Verifying an authentication assertion**  
 2610  
 2611 When verifying a given PublicKeyCredential structure (credential) as  
 2612 part of an authentication ceremony, the Relying Party MUST proceed as  
 2613 follows:  
 2614  
 2615 1. Using credential's id attribute (or the corresponding rawId, if  
 2616 base64url encoding is inappropriate for your use case), look up the  
 2617 corresponding credential public key.  
 2618  
 2619 2. Let cData, aData and sig denote the value of credential's  
 2620 response's clientDataJSON, authenticatorData, and signature  
 2621 respectively.  
 2622  
 2623 3. Perform JSON deserialization on cData to extract the client data C  
 2624 used for the signature.  
 2625  
 2626 4. Verify that the challenge member of C matches the challenge that  
 2627 was sent to the authenticator in the  
 2628 PublicKeyCredentialRequestOptions passed to the get() call.  
 2629  
 2630 5. Verify that the origin member of C matches the Relying Party's  
 2631 origin.  
 2632  
 2633 6. Verify that the tokenBindingId member of C (if present) matches the  
 2634 Token Binding ID for the TLS connection over which the signature  
 2635 was obtained.  
 2636  
 2637 7. Verify that the clientExtensions member of C is a subset of the  
 2638 extensions requested by the Relying Party and that the  
 2639 authenticatorExtensions in C is also a subset of the extensions  
 2640 requested by the Relying Party.  
 2641  
 2642 8. Verify that the RP ID hash in aData is the SHA-256 hash of the RP  
 2643 ID expected by the Relying Party.  
 2644  
 2645 9. Let hash be the result of computing a hash over the cData using the  
 2646 algorithm represented by the hashAlgorithm member of C.  
 2647  
 2648 10. Using the credential public key looked up in step 1, verify that  
 2649 sig is a valid signature over the binary concatenation of aData and  
 2650 hash.  
 2651  
 2652 11. If the signature counter value adata.signCount is nonzero or the  
 2653 value stored in conjunction with credential's id attribute is  
 2654 nonzero, then run the following substep:  
 2655 + If the signature counter value adata.signCount is  
 2656 greater than the signature counter value stored in  
 2657 conjunction with credential's id attribute,  
 2658 Update the stored signature counter value,  
 2659 associated with credential's id attribute, to be the  
 2660 value of adata.signCount.  
 2661  
 2662 less than or equal to the signature counter value stored in  
 2663 conjunction with credential's id attribute.  
 2664 This is an signal that the authenticator may be  
 2665 cloned, i.e. at least two copies of the credential  
 2666 private key may exist and are being used in  
 2667 parallel. Relying Parties should incorporate this  
 2668 information into their risk scoring. Whether the  
 2669 Relying Party updates the stored signature counter  
 2670 value in this case, or not, or fails the  
 2671 authentication ceremony or not, is Relying  
 2672 Party-specific.

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 **7.1. Attestation Statement Format Identifiers**

2504 Attestation statement formats are identified by a string, called a  
 2505 attestation statement format identifier, chosen by the author of the  
 2506 attestation statement format.

2507 Attestation statement format identifiers SHOULD be registered per  
 2508 [WebAuthn-Registries] "Registries for Web Authentication (WebAuthn)".  
 2509 All registered attestation statement format identifiers are unique  
 2510 amongst themselves as a matter of course.

2511 Unregistered attestation statement format identifiers SHOULD use  
 2512 lowercase reverse domain-name naming, using a domain name registered by  
 2513 the developer, in order to assure uniqueness of the identifier. All  
 2514 attestation statement format identifiers MUST be a maximum of 32 octets  
 2515 in length and MUST consist only of printable USASCII characters,  
 2516 excluding backslash and doublequote, i.e., VCHAR as defined in  
 2517 [RFC5234] but without %x22 and %x5c.

2518 Note: This means attestation statement format identifiers based on  
 2519 domain names MUST incorporate only LDH Labels [RFC5890].

2520 Implementations MUST match WebAuthn attestation statement format  
 2521 identifiers in a case-sensitive fashion.

2522 Attestation statement formats that may exist in multiple versions  
 2523 SHOULD include a version in their identifier. In effect, different  
 2524 versions are thus treated as different formats, e.g., packed2 as a new  
 2525 version of the packed attestation statement format.

2526 The following sections present a set of currently-defined and  
 2527 registered attestation statement formats and their identifiers. The  
 2528 up-to-date list of registered WebAuthn Extensions is maintained in the  
 2529 IANA "WebAuthn Attestation Statement Format Identifier" registry  
 2530 established by [WebAuthn-Registries].

### 2531 **7.2. Packed Attestation Statement Format**

2532 This is a WebAuthn optimized attestation statement format. It uses a  
 2533 very compact but still extensible encoding method. It is implementable  
 2534 by authenticators with limited resources (e.g., secure elements).

2535 Attestation statement format identifier  
 2536 packed

2537 Attestation types supported  
 2538 All

#### 2539 Syntax

2540 The syntax of a Packed Attestation statement is defined by the  
 2541 following CDDL:

```
2542 $$attStmtType ::= (  

  2543     fmt: "packed",  

  2544     attStmt: packedStmtFormat  

  2545 )
```

```
2546 packedStmtFormat = {  

  2547     alg: rsaAlgName / eccAlgName,  

  2548     sig: bytes,
```

2664  
 2665 **12. If all the above steps are successful, continue with the**  
 2666 authentication ceremony as appropriate. Otherwise, fail the  
 2667 authentication ceremony.  
 2668

### 2669 **8. Defined Attestation Statement Formats**

2670 WebAuthn supports pluggable attestation statement formats. This section  
 2671 defines an initial set of such formats.

### 2672 **8.1. Attestation Statement Format Identifiers**

2673 Attestation statement formats are identified by a string, called a  
 2674 attestation statement format identifier, chosen by the author of the  
 2675 attestation statement format.

2676 Attestation statement format identifiers SHOULD be registered per  
 2677 [WebAuthn-Registries] "Registries for Web Authentication (WebAuthn)".  
 2678 All registered attestation statement format identifiers are unique  
 2679 amongst themselves as a matter of course.

2680 Unregistered attestation statement format identifiers SHOULD use  
 2681 lowercase reverse domain-name naming, using a domain name registered by  
 2682 the developer, in order to assure uniqueness of the identifier. All  
 2683 attestation statement format identifiers MUST be a maximum of 32 octets  
 2684 in length and MUST consist only of printable USASCII characters,  
 2685 excluding backslash and doublequote, i.e., VCHAR as defined in  
 2686 [RFC5234] but without %x22 and %x5c.

2687 Note: This means attestation statement format identifiers based on  
 2688 domain names MUST incorporate only LDH Labels [RFC5890].

2689 Implementations MUST match WebAuthn attestation statement format  
 2690 identifiers in a case-sensitive fashion.

2691 Attestation statement formats that may exist in multiple versions  
 2692 SHOULD include a version in their identifier. In effect, different  
 2693 versions are thus treated as different formats, e.g., packed2 as a new  
 2694 version of the packed attestation statement format.

2695 The following sections present a set of currently-defined and  
 2696 registered attestation statement formats and their identifiers. The  
 2697 up-to-date list of registered WebAuthn Extensions is maintained in the  
 2698 IANA "WebAuthn Attestation Statement Format Identifier" registry  
 2699 established by [WebAuthn-Registries].

### 2700 **8.2. Packed Attestation Statement Format**

2701 This is a WebAuthn optimized attestation statement format. It uses a  
 2702 very compact but still extensible encoding method. It is implementable  
 2703 by authenticators with limited resources (e.g., secure elements).

2704 Attestation statement format identifier  
 2705 packed

2706 Attestation types supported  
 2707 All

#### 2708 Syntax

2709 The syntax of a Packed Attestation statement is defined by the  
 2710 following CDDL:

```
2711 $$attStmtType ::= (  

  2712     fmt: "packed",  

  2713     attStmt: packedStmtFormat  

  2714 )
```

```
2715 packedStmtFormat = {  

  2716     alg: COSEAlgorithmIdentifier,  

  2717     sig: bytes,
```

```

2564     x5c: [ attestnCert: bytes, * (caCert: bytes) ]
2565     } //
2566     {
2567       alg: "ED256" / "ED512",
2568
2569       sig: bytes,
2570       ecdaaKeyld: bytes
2571     }

```

The semantics of the fields are as follows:

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

**sig**  
A byte string containing the attestation signature.

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

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

#### Signing procedure

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

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

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

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

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

#### Verification procedure

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

Let `authenticatorData` denote the authenticator data claimed to

```

2734     x5c: [ attestnCert: bytes, * (caCert: bytes) ]
2735     } //
2736     {
2737       alg: COSEAlgorithmIdentifier, (-260 for ED256 / -261
2738       for ED512)
2739       sig: bytes,
2740       ecdaaKeyld: bytes
2741     }

```

The semantics of the fields are as follows:

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

**sig**  
A byte string containing the attestation signature.

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

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

#### Signing procedure

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

- Let `authenticatorData` denote the authenticator data for the attestation, and let `clientDataHash` denote the hash of the serialized client data.
- If Basic or Privacy CA attestation is in use, the authenticator produces the `sig` by concatenating `authenticatorData` and `clientDataHash`, and signing the result using an attestation private key selected through an authenticator-specific mechanism. It sets `x5c` to the certificate chain of the attestation public key and `alg` to the algorithm of the attestation private key.
- If ECDAAs is in use, the authenticator produces `sig` by concatenating `authenticatorData` and `clientDataHash`, and signing the result using ECDAAs-Sign (see section 3.5 of

[FIDOEcdaaAlgorithm]) with a ECDAAs-Issuer public key selected through an authenticator-specific mechanism (see [FIDOEcdaaAlgorithm]). It sets `alg` to the algorithm of the ECDAAs-Issuer public key and `ecdaaKeyld` to the identifier of the ECDAAs-Issuer public key (see above).

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

#### Verification procedure

The verification procedure is as follows:

- Perform CBOR decoding on the given attestation statement `attStmt` structure to obtain the attestation



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 ECDA. 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 ECDA. In**  
 2650 **this case:**  
 2651  
 2652 **+ Verify that sig is a valid signature over the concatenation of**  
 2653 **authenticatorData and clientDataHash using ECDA-Verify with**  
 2654 **ECDA-Issuer public key identified by ecdaaKeyld (see**  
 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

2797 certificate array x5c, and the signature value sig. If a  
 2798 decoding error occurs, terminate this algorithm and return an  
 2799 appropriate error.  
 2800 2. Let authenticatorData denote the authenticator data claimed to  
 2801 have been used for the attestation, and let clientDataHash  
 2802 denote the hash of the serialized client data.  
 2803 3. If x5c is present, this indicates that the attestation type is  
 2804 not ECDA. In this case:  
 2805 o Verify that sig is a valid signature over the  
 2806 concatenation of authenticatorData and clientDataHash  
 2807 using the attestation public key in x5c with the  
 2808 algorithm specified in alg.  
 2809 o Verify that x5c meets the requirements in 8.2.1 Packed  
 2810 attestation statement certificate requirements.  
 2811 o If x5c contains an extension with OID 1 3 6 1 4 1 45724 1  
 2812 1 4 (id-fido-gen-ce-aaguid) verify that the value of this  
 2813 extension matches the AAGUID in authenticatorData.  
 2814 o If successful, return attestation type Basic and trust  
 2815 path x5c.  
 2816 4. If ecdaaKeyld is present, then the attestation type is ECDA.  
 2817 In this case:  
 2818 o Verify that sig is a valid signature over the  
 2819 concatenation of authenticatorData and clientDataHash  
 2820 using ECDA-Verify with ECDA-Issuer public key  
 2821 identified by ecdaaKeyld (see [FIDOecdaaAlgorithm]).  
 2822 o If successful, return attestation type ECDA and trust  
 2823 path ecdaaKeyld.  
 2824 5. If neither x5c nor ecdaaKeyld is present, self attestation is  
 2825 in use.  
 2826 o Validate that alg matches the algorithm of the credential  
 2827 private key in authenticatorData.  
 2828 o Verify that sig is a valid signature over the  
 2829 concatenation of authenticatorData and clientDataHash  
 2830 using the credential public key with alg.  
 2831 o If successful, return attestation type Self and empty  
 2832 trust path.  
 2833  
 2834 **8.2.1. Packed attestation statement certificate requirements**  
 2835  
 2836 The attestation certificate MUST have the following fields/extensions:  
 2837 \* Version must be set to 3.  
 2838 \* Subject field MUST be set to:  
 2839  
 2840 Subject-C  
 2841 Country where the Authenticator vendor is incorporated  
 2842  
 2843 Subject-O  
 2844 Legal name of the Authenticator vendor  
 2845  
 2846 Subject-OU  
 2847 Authenticator Attestation  
 2848  
 2849 Subject-CN  
 2850 No stipulation.  
 2851  
 2852 \* If the related attestation root certificate is used for multiple  
 2853 authenticator models, the Extension OID 1 3 6 1 4 1 45724 1 1 4  
 2854 (id-fido-gen-ce-aaguid) MUST be present, containing the AAGUID as  
 2855 value.  
 2856 \* The Basic Constraints extension MUST have the CA component set to  
 2857 false  
 2858 \* An Authority Information Access (AIA) extension with entry  
 2859 id-ad-ocsp and a CRL Distribution Point extension [RFC5280] are  
 2860 both optional as the status of many attestation certificates is  
 2861 available through authenticator metadata services. See, for

269f example, the FIDO Metadata Service [FIDOMetadataService].  
 269e  
 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  `) //`  
 2725  `alg: "ED256" / "ED512",`  
 2726  `ecdaaKeyId: bytes`  
 2727  `),`  
 2728  `sig: bytes,`  
 2729  `certInfo: bytes,`  
 2730  `pubArea: bytes`  
 2731 `}`  
 2732  
 2733 The semantics of the above fields are as follows:  
 2734  
 2735 ver  
 2736 The version of the TPM specification to which the  
 2737 signature conforms.  
 2738  
 2739 alg  
 2740 [The name of the algorithm used to generate the attestation  
 2741 signature. The types `rsaAlgName` and `eccAlgName` are as  
 2742 defined in 5.3.1 Attestation data. The types "ED256" and  
 2743 "ED512" refer to the algorithms specified in  
 2744 \[FIDOEcdaaAlgorithm\].](#)  
 2745  
 2746 x5c  
 2747 The AIK certificate used for the attestation and its  
 2748 certificate chain, in X.509 encoding.  
 2749  
 2750 ecdaaKeyId  
 2751 The identifier of the ECDA-issuer public key. This is the  
 2752 BigIntegerToB encoding of the component "c" as defined  
 2753 section 3.3, step 3.5 in [FIDOEcdaaAlgorithm].  
 2754  
 2755 sig  
 2756 The attestation signature, in the form of a TPMT\_SIGNATURE  
 2757 structure as specified in [TPMv2-Part2] section 11.3.4.  
 2758  
 2759 certInfo  
 2760 The TPMS\_ATTEST structure over which the above signature  
 2761 was computed, as specified in [TPMv2-Part2] section  
 2762 10.12.8.  
 2763  
 2764 pubArea  
 2765 The TPMT\_PUBLIC structure (see [TPMv2-Part2] section  
 2766

2862 example, the FIDO Metadata Service [FIDOMetadataService].  
 2863  
 2864 **8.3. TPM Attestation Statement Format**  
 2865  
 2866 This attestation statement format is generally used by authenticators  
 2867 that use a Trusted Platform Module as their cryptographic engine.  
 2868  
 2869 Attestation statement format identifier  
 2870 tpm  
 2871  
 2872 Attestation types supported  
 2873 Privacy CA, ECDA  
 2874  
 2875 Syntax  
 2876 The syntax of a TPM Attestation statement is as follows:  
 2877  
 2878 `$$attStmtType // = (`  
 2879  `fmt: "tpm",`  
 2880  `attStmt: tpmStmtFormat`  
 2881 `)`  
 2882  
 2883 `tpmStmtFormat = {`  
 2884  `ver: "2.0",`  
 2885  `(`  
 2886  `alg: COSEAlgorithmIdentifier,`  
 2887  `x5c: [ aikCert: bytes, * (caCert: bytes) ]`  
 2888  `) //`  
 2889  `alg: COSEAlgorithmIdentifier, (-260 for ED256 / -26`  
 2890 `1 for ED512)`  
 2891  `ecdaaKeyId: bytes`  
 2892  `),`  
 2893  `sig: bytes,`  
 2894  `certInfo: bytes,`  
 2895  `pubArea: bytes`  
 2896 `}`  
 2897  
 2898 The semantics of the above fields are as follows:  
 2899  
 2900 ver  
 2901 The version of the TPM specification to which the  
 2902 signature conforms.  
 2903  
 2904 alg  
 2905 [A COSEAlgorithmIdentifier containing the identifier of the  
 2906 algorithm used to generate the attestation signature.](#)  
 2907  
 2908 x5c  
 2909 The AIK certificate used for the attestation and its  
 2910 certificate chain, in X.509 encoding.  
 2911  
 2912 ecdaaKeyId  
 2913 The identifier of the ECDA-issuer public key. This is the  
 2914 BigIntegerToB encoding of the component "c" as defined  
 2915 section 3.3, step 3.5 in [FIDOEcdaaAlgorithm].  
 2916  
 2917 sig  
 2918 The attestation signature, in the form of a TPMT\_SIGNATURE  
 2919 structure as specified in [TPMv2-Part2] section 11.3.4.  
 2920  
 2921 certInfo  
 2922 The TPMS\_ATTEST structure over which the above signature  
 2923 was computed, as specified in [TPMv2-Part2] section  
 2924 10.12.8.  
 2925  
 2926 pubArea  
 2927 The TPMT\_PUBLIC structure (see [TPMv2-Part2] section  
 2928

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.  
2794  
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 ECDA. 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 ECDA.  
2827  
2828 + Perform ECDA-Verify on sig to verify that it is a valid  
2829 signature over certInfo (see [FIDOEcdaaAlgorithm]).  
2830 + If successful, return attestation type ECDA and the  
2831 identifier of the ECDA-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.

2929 12.2.4) used by the TPM to represent the credential public  
2930 key.  
2931  
2932 **Signing procedure**  
2933 Let authenticatorData denote the authenticator data for the  
2934 attestation, and let clientDataHash denote the hash of the  
2935 serialized client data.  
2936  
2937 Concatenate authenticatorData and clientDataHash to form  
2938 attToBeSigned.  
2939  
2940 Generate a signature using the procedure specified in  
2941 [TPMv2-Part3] Section 18.2, using the attestation private key  
2942 and setting the qualifyingData parameter to attToBeSigned.  
2943  
2944 Set the pubArea field to the public area of the credential  
2945 public key, the certInfo field to the output parameter of the  
2946 same name, and the sig field to the signature obtained from the  
2947 above procedure.  
2948  
2949 **Verification procedure**  
2950 Verify that the given attestation statement is valid CBOR  
2951 conforming to the syntax defined above.  
2952  
2953 Let authenticatorData denote the authenticator data claimed to  
2954 have been used for the attestation, and let clientDataHash  
2955 denote the hash of the serialized client data.  
2956  
2957 Verify that the public key specified by the parameters and  
2958 unique fields of pubArea is identical to the public key  
2959 contained in the attestation data inside authenticatorData.  
2960  
2961 Concatenate authenticatorData and clientDataHash to form  
2962 attToBeSigned.  
2963  
2964 Validate that certInfo is valid:  
2965  
2966 + Verify that magic is set to TPM\_GENERATED\_VALUE.  
2967 + Verify that type is set to TPM\_ST\_ATTEST\_CERTIFY.  
2968 + Verify that extraData is set to attToBeSigned.  
2969 + Verify that attested contains a TPMS\_CERTIFY\_INFO structure,  
2970 whose name field contains a valid Name for pubArea, as  
2971 computed using the algorithm in the nameAlg field of pubArea  
2972 using the procedure specified in [TPMv2-Part1] section 16.  
2973  
2974 If x5c is present, this indicates that the attestation type is  
2975 not ECDA. In this case:  
2976  
2977 + Verify the sig is a valid signature over certInfo using the  
2978 attestation public key in x5c with the algorithm specified in  
2979 alg.  
2980 + Verify that x5c meets the requirements in 8.3.1 TPM  
2981 attestation statement certificate requirements.  
2982 + If x5c contains an extension with OID 1 3 6 1 4 1 45724 1 1 4  
2983 (id-fido-gen-ce-aaguid) verify that the value of this  
2984 extension matches the AAGUID in authenticatorData.  
2985 + If successful, return attestation type Privacy CA and trust  
2986 path x5c.  
2987  
2988 If ecdaaKeyId is present, then the attestation type is ECDA.  
2989  
2990 + Perform ECDA-Verify on sig to verify that it is a valid  
2991 signature over certInfo (see [FIDOEcdaaAlgorithm]).  
2992 + If successful, return attestation type ECDA and the  
2993 identifier of the ECDA-Issuer public key ecdaaKeyId.  
2994  
2995 **8.3.1. TPM attestation statement certificate requirements**  
2996  
2997 TPM attestation certificate MUST have the following fields/extensions:  
2998 \* 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].

#### 7.4. Android Key Attestation Statement Format

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

Attestation statement format identifier  
android-key

Attestation types supported  
Basic

#### Syntax

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

```

2875 $$attStmtType ::= (
2876     fmt: "android-key",
2877     attStmt: androidStmtFormat
2878 )
    
```

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:

2895  
2896  
2897

- 2999 \* Subject field MUST be set to empty.
- 3000 \* The Subject Alternative Name extension must be set as defined in
- 3001 [TPMv2-EK-Profile] section 3.2.9.
- 3002 \* The Extended Key Usage extension MUST contain the
- 3003 "joint-iso-itu-t(2) internationalorganizations(23) 133 tcg-kp(8)
- 3004 tcg-kp-AIKCertificate(3)" OID.
- 3005 \* The Basic Constraints extension MUST have the CA component set to
- 3006 false.
- 3007 \* An Authority Information Access (AIA) extension with entry
- 3008 id-ad-ocsp and a CRL Distribution Point extension [RFC5280] are
- 3009 both optional as the status of many attestation certificates is
- 3010 available through metadata services. See, for example, the FIDO
- 3011 Metadata Service [FIDOMetadataService].

#### 8.4. Android Key Attestation Statement Format

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

Attestation statement format identifier  
android-key

Attestation types supported  
Basic

#### Syntax

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

```

3037 $$attStmtType ::= (
3038     fmt: "android-key",
3039     attStmt: androidStmtFormat
3040 )
    
```

```

3042 androidStmtFormat = {
3043     alg: COSEAlgorithmIdentifier,
3044     sig: bytes,
3045     x5c: [ credCert: bytes, * (caCert: bytes) ]
3046 }
    
```

#### Signing procedure

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

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

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

#### Verification procedure

Verification is performed as follows:

3054  
3055  
3056  
3057  
3058  
3059  
3060  
3061  
3062  
3063  
3064  
3065

- 289f + Let authenticatorData denote the authenticator data claimed to
- 289e have been used for the attestation, and let clientDataHash
- 2900 denote the hash of the serialized client data.
- 2901 + Verify that the public key in the first certificate in the
- 2902 series of certificates represented by the signature matches
- 2903 the credential public key in the attestation data field of
- 2904 authenticatorData.
- 2905 + Verify that in the attestation certificate extension data:
- 2906 o The value of the attestationChallenge field is identical
- 2907 to the concatenation of authenticatorData and
- 2908 clientDataHash.
- 2909 o The AuthorizationList.allApplications field is not
- 2910 present, since PublicKeyCredentials must be bound to the
- 2911 RP ID.
- 2912 o The value in the AuthorizationList.origin field is equal
- 2913 to KM\_TAG\_GENERATED.
- 2914 o The value in the AuthorizationList.purpose field is equal
- 2915 to KM\_PURPOSE\_SIGN.
- 2916 + If successful, return attestation type Basic with the trust
- 2917 path set to the entire attestation statement.
- 2918
- 2919

### 7.5. Android SafetyNet Attestation Statement Format

2920 When the authenticator in question is a platform-provided Authenticator

2921 on certain Android platforms, the attestation statement is based on the

2922 SafetyNet API. In this case the authenticator data is completely

2923 controlled by the caller of the SafetyNet API (typically an application

2924 running on the Android platform) and the attestation statement only

2925 provides some statements about the health of the platform and the

2926 identity of the calling application.

2927 Attestation statement format identifier

2928 android-safetynet

2929 Attestation types supported

2930 Basic

2931 Syntax

2932 The syntax of an Android Attestation statement is defined as

2933 follows:

```
2934 $$attStmtType ::= (
2935     fmt: "android-safetynet",
2936     attStmt: safetynetStmtFormat
2937 )
```

```
2938 safetynetStmtFormat = {
2939     ver: text,
2940     response: bytes
2941 }
```

2942 The semantics of the above fields are as follows:

2943 ver

2944 The version number of Google Play Services responsible for

2945 providing the SafetyNet API.

2946 response

2947 The value returned by the above SafetyNet API. This value

2948 is a JWS [RFC7515] object (see SafetyNet online

2949 documentation) in Compact Serialization.

#### 2950 Signing procedure

2951 Let authenticatorData denote the authenticator data for the

2952 attestation, and let clientDataHash denote the hash of the

2953 serialized client data.

2954 Concatenate authenticatorData and clientDataHash to form

2955 attToBeSigned.

- 3066 + Let authenticatorData denote the authenticator data claimed to
- 3067 have been used for the attestation, and let clientDataHash
- 3068 denote the hash of the serialized client data.
- 3069 + Verify that the public key in the first certificate in the
- 3070 series of certificates represented by the signature matches
- 3071 the credential public key in the attestation data field of
- 3072 authenticatorData.
- 3073 + Verify that in the attestation certificate extension data:
- 3074 o The value of the attestationChallenge field is identical
- 3075 to the concatenation of authenticatorData and
- 3076 clientDataHash.
- 3077 o The AuthorizationList.allApplications field is not
- 3078 present, since PublicKeyCredentials must be bound to the
- 3079 RP ID.
- 3080 o The value in the AuthorizationList.origin field is equal
- 3081 to KM\_TAG\_GENERATED.
- 3082 o The value in the AuthorizationList.purpose field is equal
- 3083 to KM\_PURPOSE\_SIGN.
- 3084 + If successful, return attestation type Basic with the trust
- 3085 path set to the entire attestation statement.
- 3086
- 3087

### 8.5. Android SafetyNet Attestation Statement Format

3088 When the authenticator in question is a platform-provided Authenticator

3089 on certain Android platforms, the attestation statement is based on the

3090 SafetyNet API. In this case the authenticator data is completely

3091 controlled by the caller of the SafetyNet API (typically an application

3092 running on the Android platform) and the attestation statement only

3093 provides some statements about the health of the platform and the

3094 identity of the calling application.

3095 Attestation statement format identifier

3096 android-safetynet

3097 Attestation types supported

3098 Basic

3099 Syntax

3100 The syntax of an Android Attestation statement is defined as

3101 follows:

```
3102 $$attStmtType ::= (
3103     fmt: "android-safetynet",
3104     attStmt: safetynetStmtFormat
3105 )
```

```
3106 safetynetStmtFormat = {
3107     ver: text,
3108     response: bytes
3109 }
```

3110 The semantics of the above fields are as follows:

3111 ver

3112 The version number of Google Play Services responsible for

3113 providing the SafetyNet API.

3114 response

3115 The UTF-8 encoded result of the `getJwsResult()` call of the

3116 SafetyNet API. This value is a JWS [RFC7515] object (see

3117 SafetyNet online documentation) in Compact Serialization.

#### 3118 Signing procedure

3119 Let authenticatorData denote the authenticator data for the

3120 attestation, and let clientDataHash denote the hash of the

3121 serialized client data.

3122 Concatenate authenticatorData and clientDataHash to form

3123 attToBeSigned.

296E Request a SafetyNet attestation, providing attToBeSigned as the  
 296F nonce value. Set response to the result, and ver to the version  
 2970 of Google Play Services running in the authenticator.  
 2972  
 2973 Verification procedure  
 2974 Verification is performed as follows:  
 2975  
 2976 + Verify that the given attestation statement is valid CBOR  
 2977 conforming to the syntax defined above.  
 2978 + Verify that response is a valid SafetyNet response of version  
 2979 ver.  
 2980 + Verify that the nonce in the response is identical to the  
 2981 concatenation of the authenticatorData and clientDataHash.  
 2982 + Verify that the attestation certificate is issued to the  
 2983 hostname "attest.android.com" (see SafetyNet online  
 2984 documentation).  
 2985 + Verify that the ctsProfileMatch attribute in the payload of  
 2986 response is true.  
 2987 + If successful, return attestation type Basic with the trust  
 2988 path set to the above attestation certificate.  
 2989  
 2990 **7.6. FIDO U2F Attestation Statement Format**  
 2991  
 2992 This attestation statement format is used with FIDO U2F authenticators  
 2993 using the formats defined in [FIDO-U2F-Message-Formats].  
 2994  
 2995 Attestation statement format identifier  
 2996 fido-u2f  
 2997  
 2998 Attestation types supported  
 2999 Basic, self attestation  
 3000  
 3001 Syntax  
 3002 The syntax of a FIDO U2F attestation statement is defined as  
 3003 follows:  
 3004  
 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.

3136 Request a SafetyNet attestation, providing attToBeSigned as the  
 3137 nonce value. Set response to the result, and ver to the version  
 3138 of Google Play Services running in the authenticator.  
 3139  
 3140 Verification procedure  
 3141 Verification is performed as follows:  
 3142  
 3143 + Verify that the given attestation statement is valid CBOR  
 3144 conforming to the syntax defined above.  
 3145 + Verify that response is a valid SafetyNet response of version  
 3146 ver.  
 3147 + Verify that the nonce in the response is identical to the  
 3148 concatenation of the authenticatorData and clientDataHash.  
 3149 + Verify that the attestation certificate is issued to the  
 3150 hostname "attest.android.com" (see SafetyNet online  
 3151 documentation).  
 3152 + Verify that the ctsProfileMatch attribute in the payload of  
 3153 response is true.  
 3154 + If successful, return attestation type Basic with the trust  
 3155 path set to the above attestation certificate.  
 3156  
 3157 **8.6. FIDO U2F Attestation Statement Format**  
 3158  
 3159 This attestation statement format is used with FIDO U2F authenticators  
 3160 using the formats defined in [FIDO-U2F-Message-Formats].  
 3161  
 3162 Attestation statement format identifier  
 3163 fido-u2f  
 3164  
 3165 Attestation types supported  
 3166 Basic, self attestation  
 3167  
 3168 Syntax  
 3169 The syntax of a FIDO U2F attestation statement is defined as  
 3170 follows:  
 3171  
 3172 \$\$attStmtType ::= (  
 3173     fmt: "fido-u2f",  
 3174     attStmt: u2fStmtFormat  
 3175   )  
 3176  
 3177 u2fStmtFormat = {  
 3178     x5c: [ attestnCert: bytes, \* (caCert: bytes) ],  
 3179     sig: bytes  
 3180   }  
 3181  
 3182 The semantics of the above fields are as follows:  
 3183  
 3184 x5c  
 3185 The elements of this array contain the attestation  
 3186 certificate and its certificate chain, each encoded in  
 3187 X.509 format. The attestation certificate must be the  
 3188 first element in the array.  
 3189  
 3190 sig  
 3191 The attestation signature. **The signature was calculated**  
 3192 **over the (raw) U2F registration response message**  
 3193 **[FIDO-U2F-Message-Formats] received by the platform from**  
 3194 **the authenticator.**  
 3195  
 3196 Signing procedure  
 3197 If the credential public key of the given credential is not of  
 3198 algorithm -7 ("ES256"), stop and return an error. **Otherwise, let**  
 3199 **authenticatorData denote the authenticator data for the**  
 3200 **attestation, and let clientDataHash denote the hash of the**  
 3201 **serialized client data.**  
 3202  
 3203 If clientDataHash is 256 bits long, set tbsHash to this value.  
 3204



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 Verification procedure

3045 Verification is performed as follows:

- 3046 + Verify that the given attestation statement is valid CBOR  
3047 conforming to the syntax defined above.
- 3048 + If x5c is not a certificate for an ECDSA public key over the  
3049 P-256 curve, stop verification and return an error.
- 3050 + Let authenticatorData denote the authenticator data claimed to  
3051 have been used for the attestation, and let clientDataHash  
3052 denote the hash of the serialized client data.
- 3053 + If clientDataHash is 256 bits long, set tbsHash to this value.  
3054  
3055

- 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 The mechanism for generating public key credentials, as well as  
3072 requesting and generating Authentication assertions, as defined in 4  
3073 Web Authentication API, can be extended to suit particular use cases.  
3074 Each case is addressed by defining a registration extension and/or an  
3075 authentication extension.  
3076

3205 Otherwise set tbsHash to the SHA-256 hash of clientDataHash.  
 3206  
 3207 Generate a Registration Response Message as specified in  
 3208 [FIDO-U2F-Message-Formats] section 4.3, with the application  
 3209 parameter set to the SHA-256 hash of the RP ID associated with  
 3210 the given credential, the challenge parameter set to tbsHash,  
 3211 and the key handle parameter set to the credential ID of the  
 3212 given credential. Set the raw signature part of this  
 3213 Registration Response Message (i.e., without the user public  
 3214 key, key handle, and attestation certificates) as sig and set  
 3215 the attestation certificates of the attestation public key as  
 3216 x5c.

3217 Verification procedure

3218 Verification is performed as follows:

- 3219 1. Verify that the given attestation statement is valid CBOR  
3220 conforming to the syntax defined above.
- 3221 2. Perform CBOR decoding on the given attestation  
3222 statementattStmt structure to obtain the attestation  
3223 certificate array x5c, and the signature value sig. If a  
3224 decoding error occurs, terminate this algorithm and return an  
3225 appropriate error.
- 3226 3. Let attCert be value of the first element of x5c. Let  
3227 certificate public key be the public key conveyed by attCert.  
3228 If certificate public key is not an Elliptic Curve (EC) public  
3229 key over the P-256 curve, terminate this algorithm and return  
3230 an appropriate error.
- 3231 4. Let authenticatorData denote the given authenticator data  
3232 claimed to have been used for the attestation, and let  
3233 clientDataHash denote the given hash of the serialized client  
3234 data.
- 3235 5. Extract the claimed RP ID hash from authenticatorData. Extract  
3236 the claimed CredentialID and the claimed credential public key  
3237 from authenticatorData.attestation data.
- 3238 6. If clientDataHash is 256 bits long, set tbsHash to this value.  
3239 Otherwise set tbsHash to the SHA-256 hash of clientDataHash.
- 3240 7. Convert the COSE\_KEY formatted credential public key (see  
3241 Section 7 of [RFC8152]) to CTAP1/U2F public Key format  
3242 [FIDO-CTAP].
- 3243 o Let publicKeyU2F represent the result of the conversion  
3244 operation and set its first byte to 0x04. Note: This  
3245 signifies uncompressed ECC key format.
- 3246 o Extract the value corresponding to the "-2" key  
3247 (representing x coordinate) from the credential public  
3248 key, confirm its size to be of 32 bytes and concatenate  
3249 it with publicKeyU2F. If size differs or "-2" key is not  
3250 found, terminate this algorithm and return an appropriate  
3251 error.
- 3252 o Extract the value corresponding to the "-3" key  
3253 (representing y coordinate) from the credential public  
3254 key, confirm its size to be of 32 bytes and concatenate  
3255 it with publicKeyU2F. If size differs or "-3" key is not  
3256 found, terminate this algorithm and return an appropriate  
3257 error.
- 3258 8. Let verificationData be the concatenation of (0x00 ||  
3259 SHA-256(RP ID) || tbsHash || CredentialID || publicKeyU2F)  
3260 (see Section 4.3 of [FIDO-U2F-Message-Formats]).
- 3261 9. Verify the sig using verificationData and certificate public  
3262 key per [SEC1].
- 3263 10. If successful, return attestation type Basic with the trust  
3264 path set to x5c.  
3265  
3266  
3267

3268 9. WebAuthn Extensions

3269 The mechanism for generating public key credentials, as well as  
3270 requesting and generating Authentication assertions, as defined in 5  
3271 Web Authentication API, can be extended to suit particular use cases.  
3272 Each case is addressed by defining a registration extension and/or an  
3273 authentication extension.  
3274

3077  
3078  
3079  
3080  
3081  
3082  
3083  
3084  
3085  
3086  
3087  
3088  
3089  
3090  
3091  
3092  
3093  
3094  
3095  
3096  
3097  
3098  
3099  
3100  
3101  
3102  
3103  
3104  
3105  
3106  
3107  
3108  
3109  
3110  
3111  
3112  
3113  
3114  
3115  
3116  
3117  
3118  
3119  
3120  
3121  
3122  
3123  
3124  
3125  
3126  
3127  
3128  
3129  
3130  
3131  
3132  
3133  
3134  
3135  
3136  
3137  
3138  
3139  
3140  
3141  
3142  
3143  
3144  
3145  
3146

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

3275  
3276  
3277  
3278  
3279  
3280  
3281  
3282  
3283  
3284  
3285  
3286  
3287  
3288  
3289  
3290  
3291  
3292  
3293  
3294  
3295  
3296  
3297  
3298  
3299  
3300  
3301  
3302  
3303  
3304  
3305  
3306  
3307  
3308  
3309  
3310  
3311  
3312  
3313  
3314  
3315  
3316  
3317  
3318  
3319  
3320  
3321  
3322  
3323  
3324  
3325  
3326  
3327  
3328  
3329  
3330  
3331  
3332  
3333  
3334  
3335  
3336  
3337  
3338  
3339  
3340  
3341  
3342  
3343  
3344

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

3147 extension output value into JSON, provided that the CBOR output uses  
 3148 only types present in JSON.  
 3149

3150 The IANA "WebAuthn Extension Identifier" registry established by  
 3151 [WebAuthn-Registries] should be consulted for an up-to-date list of  
 3152 registered WebAuthn Extensions.  
 3153

3154 **8.1. Extension Identifiers**  
 3155

3156 Extensions are identified by a string, called an extension identifier,  
 3157 chosen by the extension author.  
 3158

3159 Extension identifiers SHOULD be registered per [WebAuthn-Registries]  
 3160 "Registries for Web Authentication (WebAuthn)". All registered  
 3161 extension identifiers are unique amongst themselves as a matter of  
 3162 course.  
 3163

3164 Unregistered extension identifiers should aim to be globally unique,  
 3165 e.g., by including the defining entity such as myCompany\_extension.  
 3166

3167 All extension identifiers MUST be a maximum of 32 octets in length and  
 3168 MUST consist only of printable USASCII characters, excluding backslash  
 3169 and doublequote, i.e., VCHAR as defined in [RFC5234] but without %x22  
 3170 and %x5c. Implementations MUST match WebAuthn extension identifiers in  
 3171 a case-sensitive fashion.  
 3172

3173 Extensions that may exist in multiple versions should take care to  
 3174 include a version in their identifier. In effect, different versions  
 3175 are thus treated as different extensions, e.g., myCompany\_extension\_01  
 3176

3177 **9** Defined Extensions defines an initial set of extensions and their  
 3178 identifiers. See the IANA "WebAuthn Extension Identifier" registry  
 3179 established by [WebAuthn-Registries] for an up-to-date list of  
 3180 registered WebAuthn Extension Identifiers.  
 3181

3182 **8.2. Defining extensions**  
 3183

3184 A definition of an extension must specify an extension identifier, a  
 3185 client extension input argument to be sent via the get() or create()  
 3186 call, the client extension processing rules, and a client extension  
 3187 output value. If the extension communicates with the authenticator  
 3188 (meaning it is an authenticator extension), it must also specify the  
 3189 CBOR authenticator extension input argument sent via the  
 3190 authenticatorGetAssertion or authenticatorMakeCredential call, the  
 3191 authenticator extension processing rules, and the CBOR authenticator  
 3192 extension output value.  
 3193

3194 Any client extension that is processed by the client MUST return a  
 3195 client extension output value so that the Relying Party knows that the  
 3196 extension was honored by the client. Similarly, any extension that  
 3197 requires authenticator processing MUST return an authenticator  
 3198 extension output to let the Relying Party know that the extension was  
 3199 honored by the authenticator. If an extension does not otherwise  
 3200 require any result values, it SHOULD be defined as returning a JSON  
 3201 Boolean client extension output result, set to true to signify that the  
 3202 extension was understood and processed. Likewise, any authenticator  
 3203 extension that does not otherwise require any result values MUST return  
 3204 a value and SHOULD return a CBOR Boolean authenticator extension output  
 3205 result, set to true to signify that the extension was understood and  
 3206 processed.  
 3207

3208 **8.3. Extending request parameters**  
 3209

3210 An extension defines one or two request arguments. The client extension  
 3211 input, which is a value that can be encoded in JSON, is passed from the  
 3212 Relying Party to the client in the get() or create() call, while the  
 3213 CBOR authenticator extension input is passed from the client to the  
 3214 authenticator for authenticator extensions during the processing of  
 3215 these calls.  
 3216

3346 extension output value into JSON, provided that the CBOR output uses  
 3347 only types present in JSON.  
 3348

3349 The IANA "WebAuthn Extension Identifier" registry established by  
 3350 [WebAuthn-Registries] should be consulted for an up-to-date list of  
 3351 registered WebAuthn Extensions.  
 3352

3353 **9.1. Extension Identifiers**  
 3354

3355 Extensions are identified by a string, called an extension identifier,  
 3356 chosen by the extension author.  
 3357

3358 Extension identifiers SHOULD be registered per [WebAuthn-Registries]  
 3359 "Registries for Web Authentication (WebAuthn)". All registered  
 3360 extension identifiers are unique amongst themselves as a matter of  
 3361 course.  
 3362

3363 Unregistered extension identifiers should aim to be globally unique,  
 3364 e.g., by including the defining entity such as myCompany\_extension.  
 3365

3366 All extension identifiers MUST be a maximum of 32 octets in length and  
 3367 MUST consist only of printable USASCII characters, excluding backslash  
 3368 and doublequote, i.e., VCHAR as defined in [RFC5234] but without %x22  
 3369 and %x5c. Implementations MUST match WebAuthn extension identifiers in  
 3370 a case-sensitive fashion.  
 3371

3372 Extensions that may exist in multiple versions should take care to  
 3373 include a version in their identifier. In effect, different versions  
 3374 are thus treated as different extensions, e.g., myCompany\_extension\_01  
 3375

3376 **10** Defined Extensions defines an initial set of extensions and their  
 3377 identifiers. See the IANA "WebAuthn Extension Identifier" registry  
 3378 established by [WebAuthn-Registries] for an up-to-date list of  
 3379 registered WebAuthn Extension Identifiers.  
 3380

3381 **9.2. Defining extensions**  
 3382

3383 A definition of an extension must specify an extension identifier, a  
 3384 client extension input argument to be sent via the get() or create()  
 3385 call, the client extension processing rules, and a client extension  
 3386 output value. If the extension communicates with the authenticator  
 3387 (meaning it is an authenticator extension), it must also specify the  
 3388 CBOR authenticator extension input argument sent via the  
 3389 authenticatorGetAssertion or authenticatorMakeCredential call, the  
 3390 authenticator extension processing rules, and the CBOR authenticator  
 3391 extension output value.  
 3392

3393 Any client extension that is processed by the client MUST return a  
 3394 client extension output value so that the Relying Party knows that the  
 3395 extension was honored by the client. Similarly, any extension that  
 3396 requires authenticator processing MUST return an authenticator  
 3397 extension output to let the Relying Party know that the extension was  
 3398 honored by the authenticator. If an extension does not otherwise  
 3399 require any result values, it SHOULD be defined as returning a JSON  
 3400 Boolean client extension output result, set to true to signify that the  
 3401 extension was understood and processed. Likewise, any authenticator  
 3402 extension that does not otherwise require any result values MUST return  
 3403 a value and SHOULD return a CBOR Boolean authenticator extension output  
 3404 result, set to true to signify that the extension was understood and  
 3405 processed.  
 3406

3407 **9.3. Extending request parameters**  
 3408

3409 An extension defines one or two request arguments. The client extension  
 3410 input, which is a value that can be encoded in JSON, is passed from the  
 3411 Relying Party to the client in the get() or create() call, while the  
 3412 CBOR authenticator extension input is passed from the client to the  
 3413 authenticator for authenticator extensions during the processing of  
 3414 these calls.



```

3217 A Relying Party simultaneously requests the use of an extension and
3218 sets its client extension input by including an entry in the extensions
3219 option to the create() or get() call. The entry key is the extension
3220 identifier and the value is the client extension input.
3221 var assertionPromise = navigator.credentials.get({
3222   publicKey: {
3223     challenge: "...",
3224     extensions: {
3225       "webauthnExample_foobar": 42
3226     }
3227   }
3228 });
3229

```

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

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

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

#### 8.4. Client extension processing

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

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

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

#### 8.5. Authenticator extension processing

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

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

The authenticator extension processing rules are used create the

```

3415 A Relying Party simultaneously requests the use of an extension and
3416 sets its client extension input by including an entry in the extensions
3417 option to the create() or get() call. The entry key is the extension
3418 identifier and the value is the client extension input.
3419 var assertionPromise = navigator.credentials.get({
3420   publicKey: {
3421     challenge: "...",
3422     extensions: {
3423       "webauthnExample_foobar": 42
3424     }
3425   }
3426 });
3427

```

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

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

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

#### 9.4. Client extension processing

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

Likewise, the client extension outputs are represented as a dictionary in the 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.

#### 9.5. Authenticator extension processing

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

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

The authenticator extension processing rules are used create the

3287 authenticator extension output from the authenticator extension input,  
3288 and possibly also other inputs, for each extension.  
3289

### 3290 8.6. Example Extension

3291 This section is not normative.

3292 To illustrate the requirements above, consider a hypothetical  
3293 registration extension and authentication extension "Geo". This  
3294 extension, if supported, enables a geolocation location to be returned  
3295 from the authenticator or client to the Relying Party.  
3296

3297 The extension identifier is chosen as webauthnExample\_geo. The client  
3298 extension input is the constant value true, since the extension does  
3299 not require the Relying Party to pass any particular information to the  
3300 client, other than that it requests the use of the extension. The  
3301 Relying Party sets this value in its request for an assertion:

```
3302 var assertionPromise =  
3303   navigator.credentials.get({  
3304     publicKey: {  
3305       challenge: "SGFuIFNvbG8gc2hvdCBmaXJzdC4",  
3306       allowCredentials: [], /* Empty filter */  
3307       extensions: { 'webauthnExample_geo': true }  
3308     }  
3309   });
```

3310 The extension also requires the client to set the authenticator  
3311 parameter to the fixed value true.

3312 The extension requires the authenticator to specify its geolocation in  
3313 the authenticator extension output, if known. The extension e.g.  
3314 specifies that the location shall be encoded as a two-element array of  
3315 floating point numbers, encoded with CBOR. An authenticator does this  
3316 by including it in the authenticator data. As an example, authenticator  
3317 data may be as follows (notation taken from [RFC7049]):

```
3318 81 (hex)           -- Flags, ED and UP both set.  
3319 20 05 58 1F       -- Signature counter  
3320 A1                -- CBOR map of one element  
3321 73                -- Key 1: CBOR text string of 19 byt  
3322 es  
3323   77 65 62 61 75 74 68 6E 45 78 61  
3324   6D 70 6C 65 5F 67 65 6F       -- "webauthnExample_geo" [=UTF-8 enc  
3325 oded=] string  
3326 82                -- Value 1: CBOR array of two elemen  
3327 ts  
3328   FA 42 82 1E B3           -- Element 1: Latitude as CBOR encod  
3329 ed float  
3330   FA C1 5F E3 7F         -- Element 2: Longitude as CBOR enco  
3331 ded float
```

3332 The extension defines the client extension output to be the geolocation  
3333 information, if known, as a GeoJSON [GeoJSON] point. The client  
3334 constructs the following client data:

```
3335 {  
3336   'extensions': {  
3337     'webauthnExample_geo': {  
3338       'type': 'Point',  
3339       'coordinates': [65.059962, -13.993041]  
3340     }  
3341   }  
3342 }
```

### 3350 9. Defined Extensions

3351 This section defines the initial set of extensions to be registered in  
3352 the IANA "WebAuthn Extension Identifier" registry established by  
3353 [WebAuthn-Registries]. These are recommended for implementation by user  
3354 agents targeting broad interoperability.  
3355

3484 authenticator extension output from the authenticator extension input,  
3485 and possibly also other inputs, for each extension.  
3486

### 3487 9.6. Example Extension

3488 This section is not normative.

3489 To illustrate the requirements above, consider a hypothetical  
3490 registration extension and authentication extension "Geo". This  
3491 extension, if supported, enables a geolocation location to be returned  
3492 from the authenticator or client to the Relying Party.  
3493

3494 The extension identifier is chosen as webauthnExample\_geo. The client  
3495 extension input is the constant value true, since the extension does  
3496 not require the Relying Party to pass any particular information to the  
3497 client, other than that it requests the use of the extension. The  
3498 Relying Party sets this value in its request for an assertion:

```
3499 var assertionPromise =  
3500   navigator.credentials.get({  
3501     publicKey: {  
3502       challenge: "SGFuIFNvbG8gc2hvdCBmaXJzdC4",  
3503       allowCredentials: [], /* Empty filter */  
3504       extensions: { 'webauthnExample_geo': true }  
3505     }  
3506   });
```

3507 The extension also requires the client to set the authenticator  
3508 parameter to the fixed value true.

3509 The extension requires the authenticator to specify its geolocation in  
3510 the authenticator extension output, if known. The extension e.g.  
3511 specifies that the location shall be encoded as a two-element array of  
3512 floating point numbers, encoded with CBOR. An authenticator does this  
3513 by including it in the authenticator data. As an example, authenticator  
3514 data may be as follows (notation taken from [RFC7049]):

```
3515 81 (hex)           -- Flags, ED and UP both set.  
3516 20 05 58 1F       -- Signature counter  
3517 A1                -- CBOR map of one element  
3518 73                -- Key 1: CBOR text string of 19 byt  
3519 es  
3520   77 65 62 61 75 74 68 6E 45 78 61  
3521   6D 70 6C 65 5F 67 65 6F       -- "webauthnExample_geo" [=UTF-8 enc  
3522 oded=] string  
3523 82                -- Value 1: CBOR array of two elemen  
3524 ts  
3525   FA 42 82 1E B3           -- Element 1: Latitude as CBOR encod  
3526 ed float  
3527   FA C1 5F E3 7F         -- Element 2: Longitude as CBOR enco  
3528 ded float
```

3529 The extension defines the client extension output to be the geolocation  
3530 information, if known, as a GeoJSON [GeoJSON] point. The client  
3531 constructs the following client data:

```
3532 {  
3533   'extensions': {  
3534     'webauthnExample_geo': {  
3535       'type': 'Point',  
3536       'coordinates': [65.059962, -13.993041]  
3537     }  
3538   }  
3539 }
```

### 3540 10. Defined Extensions

3541 This section defines the initial set of extensions to be registered in  
3542 the IANA "WebAuthn Extension Identifier" registry established by  
3543 [WebAuthn-Registries]. These are recommended for implementation by user  
3544 agents targeting broad interoperability.  
3545

3357 **9.1. FIDO AppId Extension (appid)**  
3358  
3359 This authentication extension allows Relying Parties that have  
3360 previously registered a credential using the legacy FIDO JavaScript  
3361 APIs to request an assertion. Specifically, this extension allows  
3362 Relying Parties to specify an appid [FIDO-APPID] to overwrite the  
3363 otherwise computed rpId. This extension is only valid if used during  
3364 the get() call; other usage will result in client error.  
3365  
3366 Extension identifier  
3367 appid  
3368  
3369 Client extension input  
3370 A single JSON string specifying a FIDO appid.  
3371  
3372 Client extension processing  
3373 If rpId is present, reject promise with a DOMException whose  
3374 name is "NotAllowedError", and terminate this algorithm. Replace  
3375 the calculation of rpId in Step 3 of 4.1.4 Use an existing  
3376 credential to make an assertion - **PublicKeyCredential's**  
3377 **[[DiscoverFromExternalSource]](options) method with the**  
3378 **following procedure: The client uses the value of appid to**  
3379 **perform the Appid validation procedure (as defined by**  
3380 **[FIDO-APPID]). If valid, the value of rpId for all client**  
3381 **processing should be replaced by the value of appid.**  
3382  
3383 Client extension output  
3384 Returns the JSON value true to indicate to the RP that the  
3385 extension was acted upon  
3386  
3387 Authenticator extension input  
3388 None.  
3389  
3390 Authenticator extension processing  
3391 None.  
3392  
3393 Authenticator extension output  
3394 None.  
3395  
3396 **9.2. Simple Transaction Authorization Extension (txAuthSimple)**  
3397  
3398 This registration extension and authentication extension allows for a  
3399 simple form of transaction authorization. A Relying Party can specify a  
3400 prompt string, intended for display on a trusted device on the  
3401 authenticator.  
3402  
3403 Extension identifier  
3404 txAuthSimple  
3405  
3406 Client extension input  
3407 A single JSON string prompt.  
3408  
3409 Client extension processing  
3410 None, except creating the authenticator extension input from the  
3411 client extension input.  
3412  
3413 Client extension output  
3414 Returns the authenticator extension output string UTF-8 decoded  
3415 into a JSON string  
3416  
3417 Authenticator extension input  
3418 The client extension input encoded as a CBOR text string (major  
3419 type 3).  
3420  
3421 Authenticator extension processing  
3422 The authenticator **MUST** display the prompt to the user before  
3423 performing either user verification or test of user presence.  
3424 The authenticator may insert line breaks if needed.  
3425  
3426 Authenticator extension output

3554 **10.1. FIDO AppId Extension (appid)**  
3555  
3556 This authentication extension allows Relying Parties that have  
3557 previously registered a credential using the legacy FIDO JavaScript  
3558 APIs to request an assertion. Specifically, this extension allows  
3559 Relying Parties to specify an appid [FIDO-APPID] to overwrite the  
3560 otherwise computed rpId. This extension is only valid if used during  
3561 the get() call; other usage will result in client error.  
3562  
3563 Extension identifier  
3564 appid  
3565  
3566 Client extension input  
3567 A single JSON string specifying a FIDO appid.  
3568  
3569 Client extension processing  
3570 If rpId is present, reject promise with a DOMException whose  
3571 name is "NotAllowedError", and terminate this algorithm. Replace  
3572 the calculation of rpId in Step 3 of 5.1.4 Use an existing  
3573 credential to make an assertion **with the following procedure:**  
3574 **The client uses the value of appid to perform the Appid**  
3575 **validation procedure (as defined by [FIDO-APPID]). If valid, the**  
3576 **value of rpId for all client processing should be replaced by**  
3577 **the value of appid.**  
3578  
3579 Client extension output  
3580 Returns the JSON value true to indicate to the RP that the  
3581 extension was acted upon  
3582  
3583 Authenticator extension input  
3584 None.  
3585  
3586 Authenticator extension processing  
3587 None.  
3588  
3589 Authenticator extension output  
3590 None.  
3591  
3592 **10.2. Simple Transaction Authorization Extension (txAuthSimple)**  
3593  
3594 This registration extension and authentication extension allows for a  
3595 simple form of transaction authorization. A Relying Party can specify a  
3596 prompt string, intended for display on a trusted device on the  
3597 authenticator.  
3598  
3599 Extension identifier  
3600 txAuthSimple  
3601  
3602 Client extension input  
3603 A single JSON string prompt.  
3604  
3605 Client extension processing  
3606 None, except creating the authenticator extension input from the  
3607 client extension input.  
3608  
3609 Client extension output  
3610 Returns the authenticator extension output string UTF-8 decoded  
3611 into a JSON string  
3612  
3613 Authenticator extension input  
3614 The client extension input encoded as a CBOR text string (major  
3615 type 3).  
3616  
3617 Authenticator extension processing  
3618 The authenticator **MUST** display the prompt to the user before  
3619 performing either user verification or test of user presence.  
3620 The authenticator may insert line breaks if needed.  
3621  
3622 Authenticator extension output



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

### 9.3. Generic Transaction Authorization Extension (txAuthGeneric)

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

Extension identifier  
txAuthGeneric

Client extension input  
A CBOR map defined as follows:

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

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

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

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

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

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

### 9.4. Authenticator Selection Extension (authnSel)

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

Extension identifier  
authnSel

Client extension input  
A sequence of AAGUIDs:

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

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

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

```
typedef BufferSource AAGUID;
```

Client extension processing

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

### 10.3. Generic Transaction Authorization Extension (txAuthGeneric)

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

Extension identifier  
txAuthGeneric

Client extension input  
A CBOR map defined as follows:

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

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

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

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

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

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

### 10.4. Authenticator Selection Extension (authnSel)

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

Extension identifier  
authnSel

Client extension input  
A sequence of AAGUIDs:

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

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

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

```
typedef BufferSource AAGUID;
```

Client extension processing

3497 This extension can only be used during create(). If the client  
3498 supports the Authenticator Selection Extension, it MUST use the  
3499 first available authenticator whose AAGUID is present in the  
3500 AuthenticatorSelectionList. If none of the available  
3501 authenticators match a provided AAGUID, the client MUST select  
3502 an authenticator from among the available authenticators to  
3503 generate the credential.  
3504

3505 **Client extension output**

3506 Returns the JSON value true to indicate to the RP that the  
3507 extension was acted upon  
3508

3509 **Authenticator extension input**

3510 None.  
3511

3512 **Authenticator extension processing**

3513 None.  
3514

3515 **Authenticator extension output**

3516 None.  
3517

3518 **9.5. Supported Extensions Extension (exts)**

3519 This registration extension enables the Relying Party to determine  
3520 which extensions the authenticator supports.  
3521

3522 **Extension identifier**

3523 exts  
3524

3525 **Client extension input**

3526 The Boolean value true to indicate that this extension is  
3527 requested by the Relying Party.  
3528

3529 **Client extension processing**

3530 None, except creating the authenticator extension input from the  
3531 client extension input.  
3532

3533 **Client extension output**

3534 Returns the list of supported extensions as a JSON array of  
3535 extension identifier strings  
3536

3537 **Authenticator extension input**

3538 The Boolean value true, encoded in CBOR (major type 7, value  
3539 21).  
3540

3541 **Authenticator extension processing**

3542 The authenticator sets the authenticator extension output to be  
3543 a list of extensions that the authenticator supports, as defined  
3544 below. This extension can be added to attestation objects.  
3545

3546 **Authenticator extension output**

3547 The SupportedExtensions extension is a list (CBOR array) of  
3548 extension identifier (UTF-8 encoded strings).  
3549

3550 **9.6. User Verification Index Extension (uvi)**

3551 This registration extension and authentication extension enables use of  
3552 a user verification index.  
3553

3554 **Extension identifier**

3555 uvi  
3556

3557 **Client extension input**

3558 The Boolean value true to indicate that this extension is  
3559 requested by the Relying Party.  
3560

3561 **Client extension processing**

3562 None, except creating the authenticator extension input from the  
3563 client extension input.  
3564

3565

3566

3693 This extension can only be used during create(). If the client  
3694 supports the Authenticator Selection Extension, it MUST use the  
3695 first available authenticator whose AAGUID is present in the  
3696 AuthenticatorSelectionList. If none of the available  
3697 authenticators match a provided AAGUID, the client MUST select  
3698 an authenticator from among the available authenticators to  
3699 generate the credential.  
3700

3701 **Client extension output**

3702 Returns the JSON value true to indicate to the RP that the  
3703 extension was acted upon  
3704

3705 **Authenticator extension input**

3706 None.  
3707

3708 **Authenticator extension processing**

3709 None.  
3710

3711 **Authenticator extension output**

3712 None.  
3713

3714 **10.5. Supported Extensions Extension (exts)**

3715 This registration extension enables the Relying Party to determine  
3716 which extensions the authenticator supports.  
3717

3718 **Extension identifier**

3719 exts  
3720

3721 **Client extension input**

3722 The Boolean value true to indicate that this extension is  
3723 requested by the Relying Party.  
3724

3725 **Client extension processing**

3726 None, except creating the authenticator extension input from the  
3727 client extension input.  
3728

3729 **Client extension output**

3730 Returns the list of supported extensions as a JSON array of  
3731 extension identifier strings  
3732

3733 **Authenticator extension input**

3734 The Boolean value true, encoded in CBOR (major type 7, value  
3735 21).  
3736

3737 **Authenticator extension processing**

3738 The authenticator sets the authenticator extension output to be  
3739 a list of extensions that the authenticator supports, as defined  
3740 below. This extension can be added to attestation objects.  
3741

3742 **Authenticator extension output**

3743 The SupportedExtensions extension is a list (CBOR array) of  
3744 extension identifier (UTF-8 encoded strings).  
3745

3746 **10.6. User Verification Index Extension (uvi)**

3747 This registration extension and authentication extension enables use of  
3748 a user verification index.  
3749

3750 **Extension identifier**

3751 uvi  
3752

3753 **Client extension input**

3754 The Boolean value true to indicate that this extension is  
3755 requested by the Relying Party.  
3756

3757 **Client extension processing**

3758 None, except creating the authenticator extension input from the  
3759 client extension input.  
3760

3761

3762

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 Servers supporting UVI extensions MUST support a length of up to  
 3602 32 bytes for the UVI value.

3603 Example for authenticator data containing one UVI extension

```

3604 ... -- [=RP ID=] hash (32 bytes)
3605 81 -- UP and ED set
3606 00 00 00 01 -- (initial) signature counter
3607 ... -- all public key alg etc.
3608 A1 -- extension: CBOR map of one elemen
3609 t
3610 63 -- Key 1: CBOR text string of 3 byte
3611 s
3612 75 76 69 -- "uvi" [=UTF-8 encoded=] string
3613 58 20 -- Value 1: CBOR byte string with 0x
3614 20 bytes
3615 00 43 B8 E3 BE 27 95 8C -- the UVI value itself
3616 28 D5 74 BF 46 8A 85 CF
3617 46 9A 14 F0 E5 16 69 31
3618 DA 4B CF FF C1 BB 11 32
3619 82
  
```

3620 **9.7. Location Extension (loc)**

3621 The location registration extension and authentication extension  
 3622 provides the client device's current location to the WebAuthn Relying  
 3623 Party.

3624 Extension identifier  
 3625 loc

3626 Client extension input  
 3627 The Boolean value true to indicate that this extension is  
 3628 requested by the Relying Party.

3763 Client extension output  
 3764 Returns a JSON string containing the base64url encoding of the  
 3765 authenticator extension output

3766  
 3767 Authenticator extension input  
 3768 The Boolean value true, encoded in CBOR (major type 7, value  
 3769 21).

3770  
 3771 Authenticator extension processing  
 3772 The authenticator sets the authenticator extension output to be  
 3773 a user verification index indicating the method used by the user  
 3774 to authorize the operation, as defined below. This extension can  
 3775 be added to attestation objects and assertions.

3776  
 3777 Authenticator extension output  
 3778 The user verification index (UVI) is a value uniquely  
 3779 identifying a user verification data record. The UVI is encoded  
 3780 as CBOR byte string (type 0x58). Each UVI value MUST be specific  
 3781 to the related key (in order to provide unlinkability). It also  
 3782 must contain sufficient entropy that makes guessing impractical.  
 3783 UVI values MUST NOT be reused by the Authenticator (for other  
 3784 biometric data or users).

3785  
 3786 The UVI data can be used by servers to understand whether an  
 3787 authentication was authorized by the exact same biometric data  
 3788 as the initial key generation. This allows the detection and  
 3789 prevention of "friendly fraud".

3790  
 3791 As an example, the UVI could be computed as SHA256(KeyID ||  
 3792 SHA256(rawUVI)), where || represents concatenation, and the  
 3793 rawUVI reflects (a) the biometric reference data, (b) the  
 3794 related OS level user ID and (c) an identifier which changes  
 3795 whenever a factory reset is performed for the device, e.g.  
 3796 rawUVI = biometricReferenceData || OSLevelUserID ||  
 3797 FactoryResetCounter.

3798 Servers supporting UVI extensions MUST support a length of up to  
 3799 32 bytes for the UVI value.

3800 Example for authenticator data containing one UVI extension

```

3801 ... -- [=RP ID=] hash (32 bytes)
3802 81 -- UP and ED set
3803 00 00 00 01 -- (initial) signature counter
3804 ... -- all public key alg etc.
3805 A1 -- extension: CBOR map of one elemen
3806 t
3807 63 -- Key 1: CBOR text string of 3 byte
3808 s
3809 75 76 69 -- "uvi" [=UTF-8 encoded=] string
3810 58 20 -- Value 1: CBOR byte string with 0x
3811 20 bytes
3812 00 43 B8 E3 BE 27 95 8C -- the UVI value itself
3813 28 D5 74 BF 46 8A 85 CF
3814 46 9A 14 F0 E5 16 69 31
3815 DA 4B CF FF C1 BB 11 32
3816 82
  
```

3817 **10.7. Location Extension (loc)**

3818 The location registration extension and authentication extension  
 3819 provides the client device's current location to the WebAuthn Relying  
 3820 Party.

3821 Extension identifier  
 3822 loc

3823 Client extension input  
 3824 The Boolean value true to indicate that this extension is  
 3825 requested by the Relying Party.



```

3636 Client extension processing
3637   None, except creating the authenticator extension input from the
3638   client extension input.
3639
3640 Client extension output
3641   Returns a JSON object that encodes the location information in
3642   the authenticator extension output as a Coordinates value, as
3643   defined by The W3C Geolocation API Specification.
3644
3645 Authenticator extension input
3646   The Boolean value true, encoded in CBOR (major type 7, value
3647   21).
3648
3649 Authenticator extension processing
3650   If the authenticator does not support the extension, then the
3651   authenticator MUST ignore the extension request. If the
3652   authenticator accepts the extension, then the authenticator
3653   SHOULD only add this extension data to a packed attestation or
3654   assertion.
3655
3656 Authenticator extension output
3657   If the authenticator accepts the extension request, then
3658   authenticator extension output SHOULD provide location data in
3659   the form of a CBOR-encoded map, with the first value being the
3660   extension identifier and the second being an array of returned
3661   values. The array elements SHOULD be derived from (key,value)
3662   pairings for each location attribute that the authenticator
3663   supports. The following is an example of authenticator data
3664   where the returned array is comprised of a {longitude, latitude,
3665   altitude} triplet, following the coordinate representation
3666   defined in The W3C Geolocation API Specification.
3667
3668   ... -- [=RP ID=] hash (32 bytes)
3669   81 -- UP and ED set
3670   00 00 00 01 -- (initial) signature counter
3671   ... -- all public key alg etc.
3672   A1 -- extension: CBOR map of one elemen
3673   t
3674   63 -- Value 1: CBOR text string of 3 by
3675   tes
3676   6C 6F 63 -- "loc" [=UTF-8 encoded=] string
3677   86 -- Value 2: array of 6 elements
3678   68 -- Element 1: CBOR text string of 8 bytes
3679   6C 61 74 69 74 75 64 65 -- "latitude" [=UTF-8 encoded=] stri
3680   ng
3681   FB ... -- Element 2: Latitude as CBOR encoded double-p
3682   recision float
3683   69 -- Element 3: CBOR text string of 9 bytes
3684   6C 6F 6E 67 69 74 75 64 65 -- "longitude" [=UTF-8 encoded=] str
3685   ing
3686   FB ... -- Element 4: Longitude as CBOR encoded double-
3687   precision float
3688   68 -- Element 5: CBOR text string of 8 bytes
3689   61 6C 74 69 74 75 64 65 -- "altitude" [=UTF-8 encoded=] stri
3690   ng
3691   FB ... -- Element 6: Altitude as CBOR encoded double-p
3692   recision float
3693
3694 9.8. User Verification Method Extension (uvm)
3695
3696   This registration extension and authentication extension enables use of
3697   a user verification method.
3698
3699   Extension identifier
3700   uvm
3701
3702 Client extension input
3703   The Boolean value true to indicate that this extension is
3704   requested by the WebAuthn Relying Party.
3705

```

```

3833 Client extension processing
3834   None, except creating the authenticator extension input from the
3835   client extension input.
3836
3837 Client extension output
3838   Returns a JSON object that encodes the location information in
3839   the authenticator extension output as a Coordinates value, as
3840   defined by The W3C Geolocation API Specification.
3841
3842 Authenticator extension input
3843   The Boolean value true, encoded in CBOR (major type 7, value
3844   21).
3845
3846 Authenticator extension processing
3847   If the authenticator does not support the extension, then the
3848   authenticator MUST ignore the extension request. If the
3849   authenticator accepts the extension, then the authenticator
3850   SHOULD only add this extension data to a packed attestation or
3851   assertion.
3852
3853 Authenticator extension output
3854   If the authenticator accepts the extension request, then
3855   authenticator extension output SHOULD provide location data in
3856   the form of a CBOR-encoded map, with the first value being the
3857   extension identifier and the second being an array of returned
3858   values. The array elements SHOULD be derived from (key,value)
3859   pairings for each location attribute that the authenticator
3860   supports. The following is an example of authenticator data
3861   where the returned array is comprised of a {longitude, latitude,
3862   altitude} triplet, following the coordinate representation
3863   defined in The W3C Geolocation API Specification.
3864
3865   ... -- [=RP ID=] hash (32 bytes)
3866   81 -- UP and ED set
3867   00 00 00 01 -- (initial) signature counter
3868   ... -- all public key alg etc.
3869   A1 -- extension: CBOR map of one elemen
3870   t
3871   63 -- Value 1: CBOR text string of 3 by
3872   tes
3873   6C 6F 63 -- "loc" [=UTF-8 encoded=] string
3874   86 -- Value 2: array of 6 elements
3875   68 -- Element 1: CBOR text string of 8 bytes
3876   6C 61 74 69 74 75 64 65 -- "latitude" [=UTF-8 encoded=] stri
3877   ng
3878   FB ... -- Element 2: Latitude as CBOR encoded double-p
3879   recision float
3880   69 -- Element 3: CBOR text string of 9 bytes
3881   6C 6F 6E 67 69 74 75 64 65 -- "longitude" [=UTF-8 encoded=] str
3882   ing
3883   FB ... -- Element 4: Longitude as CBOR encoded double-
3884   precision float
3885   68 -- Element 5: CBOR text string of 8 bytes
3886   61 6C 74 69 74 75 64 65 -- "altitude" [=UTF-8 encoded=] stri
3887   ng
3888   FB ... -- Element 6: Altitude as CBOR encoded double-p
3889   recision float
3890
3891 10.8. User Verification Method Extension (uvm)
3892
3893   This registration extension and authentication extension enables use of
3894   a user verification method.
3895
3896   Extension identifier
3897   uvm
3898
3899 Client extension input
3900   The Boolean value true to indicate that this extension is
3901   requested by the WebAuthn Relying Party.
3902

```

3706 Client extension processing  
 3707 None, except creating the authenticator extension input from the  
 3708 client extension input.  
 3709  
 3710 Client extension output  
 3711 Returns a JSON array of 3-element arrays of numbers that encodes  
 3712 the factors in the authenticator extension output  
 3713  
 3714 Authenticator extension input  
 3715 The Boolean value true, encoded in CBOR (major type 7, value  
 3716 21).  
 3717  
 3718 Authenticator extension processing  
 3719 The authenticator sets the authenticator extension output to be  
 3720 a user verification index indicating the method used by the user  
 3721 to authorize the operation, as defined below. This extension can  
 3722 be added to attestation objects and assertions.  
 3723  
 3724 Authenticator extension output  
 3725 Authenticators can report up to 3 different user verification  
 3726 methods (factors) used in a single authentication instance,  
 3727 using the CBOR syntax defined below:  
 3728  
 3729 uvmFormat = [ 1\*3 uvmEntry ]  
 3730 uvmEntry = [  
 3731     userVerificationMethod: uint .size 4,  
 3732     keyProtectionType: uint .size 2,  
 3733     matcherProtectionType: uint .size 2  
 3734     ]  
 3735  
 3736 The semantics of the fields in each uvmEntry are as follows:  
 3737  
 3738 userVerificationMethod  
 3739 The authentication method/factor used by the authenticator  
 3740 to verify the user. Available values are defined in  
 3741 [FIDOReg], "User Verification Methods" section.  
 3742  
 3743 keyProtectionType  
 3744 The method used by the authenticator to protect the FIDO  
 3745 registration private key material. Available values are  
 3746 defined in [FIDOReg], "Key Protection Types" section.  
 3747  
 3748 matcherProtectionType  
 3749 The method used by the authenticator to protect the  
 3750 matcher that performs user verification. Available values  
 3751 are defined in [FIDOReg], "Matcher Protection Types"  
 3752 section.  
 3753  
 3754 If >3 factors can be used in an authentication instance the  
 3755 authenticator vendor must select the 3 factors it believes will  
 3756 be most relevant to the Server to include in the UVM.  
 3757  
 3758 Example for authenticator data containing one UVM extension for  
 3759 a multi-factor authentication instance where 2 factors were  
 3760 used:  
 3761  
 3762  
 3763 ... -- [=RP ID=] hash (32 bytes)  
 3764 81 -- UP and ED set  
 3765 00 00 00 01 -- (initial) signature counter  
 3766 ... -- all public key alg etc.  
 3767 A1 -- extension: CBOR map of one element  
 3768 63 -- Key 1: CBOR text string of 3 bytes  
 3769 75 76 6d -- "uvm" [=UTF-8 encoded=] string  
 3770 82 -- Value 1: CBOR array of length 2 indicating two factor  
 3771 usage  
 3772 83 -- Item 1: CBOR array of length 3  
 3773 02 -- Subitem 1: CBOR integer for User Verification Method  
 3774 Fingerprint  
 3775 04 -- Subitem 2: CBOR short for Key Protection Type TEE

3903 Client extension processing  
 3904 None, except creating the authenticator extension input from the  
 3905 client extension input.  
 3906  
 3907 Client extension output  
 3908 Returns a JSON array of 3-element arrays of numbers that encodes  
 3909 the factors in the authenticator extension output  
 3910  
 3911 Authenticator extension input  
 3912 The Boolean value true, encoded in CBOR (major type 7, value  
 3913 21).  
 3914  
 3915 Authenticator extension processing  
 3916 The authenticator sets the authenticator extension output to be  
 3917 a user verification index indicating the method used by the user  
 3918 to authorize the operation, as defined below. This extension can  
 3919 be added to attestation objects and assertions.  
 3920  
 3921 Authenticator extension output  
 3922 Authenticators can report up to 3 different user verification  
 3923 methods (factors) used in a single authentication instance,  
 3924 using the CBOR syntax defined below:  
 3925  
 3926 uvmFormat = [ 1\*3 uvmEntry ]  
 3927 uvmEntry = [  
 3928     userVerificationMethod: uint .size 4,  
 3929     keyProtectionType: uint .size 2,  
 3930     matcherProtectionType: uint .size 2  
 3931     ]  
 3932  
 3933 The semantics of the fields in each uvmEntry are as follows:  
 3934  
 3935 userVerificationMethod  
 3936 The authentication method/factor used by the authenticator  
 3937 to verify the user. Available values are defined in  
 3938 [FIDOReg], "User Verification Methods" section.  
 3939  
 3940 keyProtectionType  
 3941 The method used by the authenticator to protect the FIDO  
 3942 registration private key material. Available values are  
 3943 defined in [FIDOReg], "Key Protection Types" section.  
 3944  
 3945 matcherProtectionType  
 3946 The method used by the authenticator to protect the  
 3947 matcher that performs user verification. Available values  
 3948 are defined in [FIDOReg], "Matcher Protection Types"  
 3949 section.  
 3950  
 3951 If >3 factors can be used in an authentication instance the  
 3952 authenticator vendor must select the 3 factors it believes will  
 3953 be most relevant to the Server to include in the UVM.  
 3954  
 3955 Example for authenticator data containing one UVM extension for  
 3956 a multi-factor authentication instance where 2 factors were  
 3957 used:  
 3958  
 3959  
 3960 ... -- [=RP ID=] hash (32 bytes)  
 3961 81 -- UP and ED set  
 3962 00 00 00 01 -- (initial) signature counter  
 3963 ... -- all public key alg etc.  
 3964 A1 -- extension: CBOR map of one element  
 3965 63 -- Key 1: CBOR text string of 3 bytes  
 3966 75 76 6d -- "uvm" [=UTF-8 encoded=] string  
 3967 82 -- Value 1: CBOR array of length 2 indicating two factor  
 3968 usage  
 3969 83 -- Item 1: CBOR array of length 3  
 3970 02 -- Subitem 1: CBOR integer for User Verification Method  
 3971 Fingerprint  
 3972 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

```

## 10. IANA Considerations

### 10.1. WebAuthn Attestation Statement Format Identifier Registrations

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

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

### 10.2. WebAuthn Extension Identifier Registrations

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

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

```

3973     02 -- Subitem 3: CBOR short for Matcher Protection Type TE
3974 E
3975     83 -- Item 2: CBOR array of length 3
3976     04 -- Subitem 1: CBOR integer for User Verification Method
3977 Passcode
3978     01 -- Subitem 2: CBOR short for Key Protection Type Softwa
3979 re
3980     01 -- Subitem 3: CBOR short for Matcher Protection Type So
3981 ftware
3982

```

## 11. IANA Considerations

### 11.1. WebAuthn Attestation Statement Format Identifier Registrations

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

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

### 11.2. WebAuthn Extension Identifier Registrations

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

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



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

### 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].

3901 \* Name: RS256  
 3902 \* Value: -257  
 3903 \* Description: RSASSA-PKCS1-v1\_5 w/ SHA-256  
 3904 \* Reference: Section 8.2 of [RFC8017]  
 3905 \* Recommended: No  
 3906 \* Name: RS384  
 3907 \* Value: -258  
 3908 \* Description: RSASSA-PKCS1-v1\_5 w/ SHA-384  
 3909 \* Reference: Section 8.2 of [RFC8017]  
 3910 \* Recommended: No  
 3911 \* Name: RS512  
 3912 \* Value: -259  
 3913 \* Description: RSASSA-PKCS1-v1\_5 w/ SHA-512  
 3914 \* Reference: Section 8.2 of [RFC8017]  
 3915 \* Recommended: No

4043 extension allows images to be used as transaction authorization  
 4044 prompts as well. This allows authenticators without a font  
 4045 rendering engine to be used and also supports a richer visual  
 4046 appearance than accomplished with the webauthn.txauth.simple  
 4047 extension.  
 4048 \* Specification Document: Section 10.3 Generic Transaction  
 4049 Authorization Extension (txAuthGeneric) of this specification  
 4050 \* WebAuthn Extension Identifier: authnSel  
 4051 \* Description: This registration extension allows a WebAuthn Relying  
 4052 Party to guide the selection of the authenticator that will be  
 4053 leveraged when creating the credential. It is intended primarily  
 4054 for WebAuthn Relying Parties that wish to tightly control the  
 4055 experience around credential creation.  
 4056 \* Specification Document: Section 10.4 Authenticator Selection  
 4057 Extension (authnSel) of this specification  
 4058 \* WebAuthn Extension Identifier: exts  
 4059 \* Description: This registration extension enables the Relying Party  
 4060 to determine which extensions the authenticator supports. The  
 4061 extension data is a list (CBOR array) of extension identifiers  
 4062 encoded as UTF-8 Strings. This extension is added automatically by  
 4063 the authenticator. This extension can be added to attestation  
 4064 statements.  
 4065 \* Specification Document: Section 10.5 Supported Extensions  
 4066 Extension (exts) of this specification  
 4067 \* WebAuthn Extension Identifier: uvi  
 4068 \* Description: This registration extension and authentication  
 4069 extension enables use of a user verification index. The user  
 4070 verification index is a value uniquely identifying a user  
 4071 verification data record. The UVI data can be used by servers to  
 4072 understand whether an authentication was authorized by the exact  
 4073 same biometric data as the initial key generation. This allows the  
 4074 detection and prevention of "friendly fraud".  
 4075 \* Specification Document: Section 10.6 User Verification Index  
 4076 Extension (uvi) of this specification  
 4077 \* WebAuthn Extension Identifier: loc  
 4078 \* Description: The location registration extension and authentication  
 4079 extension provides the client device's current location to the  
 4080 WebAuthn relying party, if supported by the client device and  
 4081 subject to user consent.  
 4082 \* Specification Document: Section 10.7 Location Extension (loc) of  
 4083 this specification  
 4084 \* WebAuthn Extension Identifier: uvm  
 4085 \* Description: This registration extension and authentication  
 4086 extension enables use of a user verification method. The user  
 4087 verification method extension returns to the Webauthn relying party  
 4088 which user verification methods (factors) were used for the  
 4089 WebAuthn operation.  
 4090 \* Specification Document: Section 10.8 User Verification Method  
 4091 Extension (uvm) of this specification

### 11.3. 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].

4092 \* Name: RS256  
 4093 \* Value: -257  
 4094 \* Description: RSASSA-PKCS1-v1\_5 w/ SHA-256  
 4095 \* Reference: Section 8.2 of [RFC8017]  
 4096 \* Recommended: No  
 4097 \* Name: RS384  
 4098 \* Value: -258  
 4099 \* Description: RSASSA-PKCS1-v1\_5 w/ SHA-384  
 4100 \* Reference: Section 8.2 of [RFC8017]  
 4101 \* Recommended: No  
 4102 \* Name: RS512  
 4103 \* Value: -259  
 4104 \* Description: RSASSA-PKCS1-v1\_5 w/ SHA-512  
 4105 \* Reference: Section 8.2 of [RFC8017]  
 4106 \* Recommended: No

3916  
3917  
3918  
3919  
3920  
3921  
3922  
3923  
3924  
3925  
3926  
3927  
3928  
3929  
3930  
3931  
3932  
3933  
3934  
3935  
3936  
3937  
3938  
3939  
3940  
3941  
3942  
3943  
3944  
3945  
3946  
3947  
3948  
3949  
3950  
3951  
3952  
3953  
3954  
3955  
3956  
3957  
3958  
3959  
3960  
3961  
3962  
3963  
3964  
3965  
3966  
3967  
3968  
3969  
3970  
3971  
3972  
3973  
3974  
3975

## 11. Sample scenarios

This section is not normative.

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

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

### 11.1. Registration

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

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

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

4113  
4114  
4115  
4116  
4117  
4118  
4119  
4120  
4121  
4122  
4123  
4124  
4125  
4126  
4127  
4128  
4129  
4130  
4131  
4132  
4133  
4134  
4135  
4136  
4137  
4138  
4139  
4140  
4141  
4142  
4143  
4144  
4145  
4146  
4147  
4148  
4149  
4150  
4151  
4152  
4153  
4154  
4155  
4156  
4157  
4158  
4159  
4160  
4161  
4162  
4163  
4164  
4165  
4166  
4167  
4168  
4169  
4170  
4171  
4172  
4173  
4174  
4175  
4176  
4177  
4178  
4179  
4180  
4181  
4182

- \* Name: ED256
- \* Value: -260
- \* Description: TPM\_ECC\_BN\_P256 curve w/ SHA-256
- \* Reference: Section 4.2 of [FIDOEcdaaAlgorithm]
- \* Recommended: Yes
- \* Name: ED512
- \* Value: -261
- \* Description: ECC\_BN\_ISOP512 curve w/ SHA-512
- \* Reference: Section 4.2 of [FIDOEcdaaAlgorithm]
- \* Recommended: Yes

## 12. Sample scenarios

This section is not normative.

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

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

### 12.1. Registration

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

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

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

```

3976 var publicKey = {
3977   challenge: Uint8Array.from(window.atob("PGifxAoBwCkWkm4b1Cill5otCphilh6MijdbjW
3978   FjomA="), c=>c.charCodeAt(0)),
3980   // Relying Party:
3981   rp: {
3982     name: "Acme"
3983   },
3984   // User:
3985   user: {
3986     id: "1098237235409872"
3987     name: "john.p.smith@example.com",
3988     displayName: "John P. Smith",
3989     icon: "https://pics.acme.com/00/p/aBjjjpqPb.png"
3990   },
3991   // This Relying Party will accept either an ES256 or RS256 credential, but
3992   // prefers an ES256 credential.
3993   pubKeyCredParams: [
3994     {
3995       type: "public-key",
3996       alg: -7 // "ES256" as registered in the IANA COSE Algorithms registry
3997     },
3998     {
3999       type: "public-key",
4000       alg: -257 // Value registered by this specification for "RS256"
4001     }
4002   ],
4003   timeout: 60000, // 1 minute
4004   excludeCredentials: [], // No exclude list of PKCredDescriptors
4005   extensions: {"webauthn.location": true} // Include location information
4006   // in attestation
4007 };
4008 // Note: The following call will cause the authenticator to display UI.
4009 navigator.credentials.create({ publicKey })
4010 .then(function (newCredentialInfo) {
4011   // Send new credential info to server for verification and registration.
4012   // No acceptable authenticator or user refused consent. Handle appropriately
4013   // No acceptable authenticator or user refused consent. Handle appropriately
4014   // No acceptable authenticator or user refused consent. Handle appropriately
4015   // No acceptable authenticator or user refused consent. Handle appropriately
4016   // No acceptable authenticator or user refused consent. Handle appropriately
4017   // No acceptable authenticator or user refused consent. Handle appropriately
4018   // No acceptable authenticator or user refused consent. Handle appropriately
4019   // No acceptable authenticator or user refused consent. Handle appropriately
4020   // No acceptable authenticator or user refused consent. Handle appropriately
4021   // No acceptable authenticator or user refused consent. Handle appropriately
4022   // No acceptable authenticator or user refused consent. Handle appropriately
4023   // No acceptable authenticator or user refused consent. Handle appropriately
4024   // No acceptable authenticator or user refused consent. Handle appropriately
4025   // No acceptable authenticator or user refused consent. Handle appropriately
4026   // No acceptable authenticator or user refused consent. Handle appropriately
4027   // No acceptable authenticator or user refused consent. Handle appropriately
4028   // No acceptable authenticator or user refused consent. Handle appropriately
4029   // No acceptable authenticator or user refused consent. Handle appropriately
4030   // No acceptable authenticator or user refused consent. Handle appropriately
4031   // No acceptable authenticator or user refused consent. Handle appropriately
4032   // No acceptable authenticator or user refused consent. Handle appropriately
4033   // No acceptable authenticator or user refused consent. Handle appropriately
4034   // No acceptable authenticator or user refused consent. Handle appropriately
4035   // No acceptable authenticator or user refused consent. Handle appropriately
4036   // No acceptable authenticator or user refused consent. Handle appropriately
4037   // No acceptable authenticator or user refused consent. Handle appropriately
4038   // No acceptable authenticator or user refused consent. Handle appropriately
4039   // No acceptable authenticator or user refused consent. Handle appropriately
4040   // No acceptable authenticator or user refused consent. Handle appropriately
4041   // No acceptable authenticator or user refused consent. Handle appropriately
4042   // No acceptable authenticator or user refused consent. Handle appropriately
4043   // No acceptable authenticator or user refused consent. Handle appropriately
4044   // No acceptable authenticator or user refused consent. Handle appropriately
4045   // No acceptable authenticator or user refused consent. Handle appropriately

```

```

4183 var publicKey = {
4184   challenge: Uint8Array.from(window.atob("PGifxAoBwCkWkm4b1Cill5otCphilh6MijdbjW
4185   FjomA="), c=>c.charCodeAt(0)),
4187   // Relying Party:
4188   rp: {
4189     name: "Acme"
4190   },
4191   // User:
4192   user: {
4193     id: "1098237235409872",
4194     name: "john.p.smith@example.com",
4195     displayName: "John P. Smith",
4196     icon: "https://pics.acme.com/00/p/aBjjjpqPb.png"
4197   },
4198   // This Relying Party will accept either an ES256 or RS256 credential, but
4199   // prefers an ES256 credential.
4200   pubKeyCredParams: [
4201     {
4202       type: "public-key",
4203       alg: -7 // "ES256" as registered in the IANA COSE Algorithms registry
4204     },
4205     {
4206       type: "public-key",
4207       alg: -257 // Value registered by this specification for "RS256"
4208     }
4209   ],
4210   timeout: 60000, // 1 minute
4211   excludeCredentials: [], // No exclude list of PKCredDescriptors
4212   extensions: {"webauthn.location": true} // Include location information
4213   // in attestation
4214 };
4215 // Note: The following call will cause the authenticator to display UI.
4216 navigator.credentials.create({ publicKey })
4217 .then(function (newCredentialInfo) {
4218   // Send new credential info to server for verification and registration.
4219   // No acceptable authenticator or user refused consent. Handle appropriately
4220   // No acceptable authenticator or user refused consent. Handle appropriately
4221   // No acceptable authenticator or user refused consent. Handle appropriately
4222   // No acceptable authenticator or user refused consent. Handle appropriately
4223   // No acceptable authenticator or user refused consent. Handle appropriately
4224   // No acceptable authenticator or user refused consent. Handle appropriately
4225   // No acceptable authenticator or user refused consent. Handle appropriately
4226   // No acceptable authenticator or user refused consent. Handle appropriately
4227   // No acceptable authenticator or user refused consent. Handle appropriately
4228   // No acceptable authenticator or user refused consent. Handle appropriately
4229   // No acceptable authenticator or user refused consent. Handle appropriately
4230   // No acceptable authenticator or user refused consent. Handle appropriately
4231   // No acceptable authenticator or user refused consent. Handle appropriately
4232   // No acceptable authenticator or user refused consent. Handle appropriately
4233   // No acceptable authenticator or user refused consent. Handle appropriately
4234   // No acceptable authenticator or user refused consent. Handle appropriately
4235   // No acceptable authenticator or user refused consent. Handle appropriately
4236   // No acceptable authenticator or user refused consent. Handle appropriately
4237   // No acceptable authenticator or user refused consent. Handle appropriately
4238   // No acceptable authenticator or user refused consent. Handle appropriately
4239   // No acceptable authenticator or user refused consent. Handle appropriately
4240   // No acceptable authenticator or user refused consent. Handle appropriately
4241   // No acceptable authenticator or user refused consent. Handle appropriately
4242   // No acceptable authenticator or user refused consent. Handle appropriately
4243   // No acceptable authenticator or user refused consent. Handle appropriately
4244   // No acceptable authenticator or user refused consent. Handle appropriately
4245   // No acceptable authenticator or user refused consent. Handle appropriately
4246   // No acceptable authenticator or user refused consent. Handle appropriately
4247   // No acceptable authenticator or user refused consent. Handle appropriately
4248   // No acceptable authenticator or user refused consent. Handle appropriately
4249   // No acceptable authenticator or user refused consent. Handle appropriately
4250   // No acceptable authenticator or user refused consent. Handle appropriately
4251   // No acceptable authenticator or user refused consent. Handle appropriately
4252   // No acceptable authenticator or user refused consent. Handle appropriately

```



```

4046     if (userIntent) {
4047         var publicKeyOptions = { /* Public key credential creation options.
4048     */};
4049
4050         // Create and register credentials.
4051         return navigator.credentials.create({ "publicKey": publicKeyOptions
4052     });
4053     } else {
4054
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     });

```

### 11.3. Authentication

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

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

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

```

4110 if (!PublicKeyCredential) { /* Platform not capable. Handle error. */ }
4111
4112 var options = {
4113     challenge: new TextEncoder().encode("climb a mountain"),
4114     timeout: 60000, // 1 minute
4115     allowCredentials: [{ type: "public-key" }]

```

```

4253     if (userIntent) {
4254         var publicKeyOptions = { /* Public key credential creation options.
4255     */};
4256
4257         // Create and register credentials.
4258         return navigator.credentials.create({ "publicKey": publicKeyOptions
4259     });
4260     } else {
4261
4262         // Record that the user does not intend to use a platform authentica
4263     tor
4264         // and default the user to a password-based flow in the future.
4265     }
4266
4267     }).then(function (newCredentialInfo) {
4268         // Send new credential info to server for verification and registration.
4269     }).catch( function(err) {
4270         // Something went wrong. Handle appropriately.
4271     });

```

### 12.3. Authentication

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

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

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

```

4317 if (!PublicKeyCredential) { /* Platform not capable. Handle error. */ }
4318
4319 var options = {
4320     challenge: new TextEncoder().encode("climb a mountain"),
4321     timeout: 60000, // 1 minute
4322     allowCredentials: [{ type: "public-key" }]

```

```

4116     };
4117
4118     navigator.credentials.get({ "publicKey": options })
4119     .then(function (assertion) {
4120     // Send assertion to server for verification
4121     }).catch(function (err) {
4122     // No acceptable credential or user refused consent. Handle appropriately.
4123     });
4124
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     extensions: { 'webauthn.txauth.simple':
4148     "Wave your hands in the air like you just don't care" };
4149     };
4150
4151     navigator.credentials.get({ "publicKey": options })
4152     .then(function (assertion) {
4153     // Send assertion to server for verification
4154     }).catch(function (err) {
4155     // No acceptable credential or user refused consent. Handle appropriately.
4156     });
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.

```

4323     };
4324
4325     navigator.credentials.get({ "publicKey": options })
4326     .then(function (assertion) {
4327     // Send assertion to server for verification
4328     }).catch(function (err) {
4329     // No acceptable credential or user refused consent. Handle appropriately.
4330     });
4331
4332     On the other hand, if the Relying Party script has some hints to help
4333     it narrow the list of credentials, then the sample code for performing
4334     such an authentication might look like the following. Note that this
4335     sample also demonstrates how to use the extension for transaction
4336     authorization.
4337     if (!PublicKeyCredential) { /* Platform not capable. Handle error. */ }
4338
4339     var encoder = new TextEncoder();
4340     var acceptableCredential1 = {
4341     type: "public-key",
4342     id: encoder.encode("!!!!!!!hi there!!!!!!!\n")
4343     };
4344     var acceptableCredential2 = {
4345     type: "public-key",
4346     id: encoder.encode("roses are red, violets are blue\n")
4347     };
4348
4349     var options = {
4350     challenge: encoder.encode("climb a mountain"),
4351     timeout: 60000, // 1 minute
4352     allowCredentials: [acceptableCredential1, acceptableCredential2]
4353     };
4354     extensions: { 'webauthn.txauth.simple':
4355     "Wave your hands in the air like you just don't care" };
4356     };
4357
4358     navigator.credentials.get({ "publicKey": options })
4359     .then(function (assertion) {
4360     // Send assertion to server for verification
4361     }).catch(function (err) {
4362     // No acceptable credential or user refused consent. Handle appropriately.
4363     });
4364

```

#### 12.4. 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.

4186 + Sometime later, the server deregisters this credential due to  
4187 inactivity.  
4188

## 4185 12. Acknowledgements

4190 We thank the following for their contributions to, and thorough review  
4191 of, this specification: Richard Barnes, Dominic Battr, Domenic  
4192 Denicola, Rahul Ghosh, Brad Hill, Jing Jin, Angelo Liao, Anne van  
4193 Kesteren, Ian Kilpatrick, Giridhar Mandyam, Axel Nennker, Kimberly  
4194 Paulhamus, Adam Powers, Yaron Sheffer, Mike West, Jeffrey Yasskin,  
4195 Boris Zbarsky.  
4196

## 4196 Index

### 4196 Terms defined by this specification

- 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 \* [Attestation](#), in 3
- 4210 \* [Attestation Certificate](#), in 3
- 4211 \* [Attestation data](#), in 5.3.1
- 4212 \* [attestation key pair](#), in 3
- 4213 \* [attestationObject](#), in 4.2.1
- 4214 \* [attestation object](#), in 5.3
- 4215 \* [attestation private key](#), in 3
- 4216 \* [attestation public key](#), in 3
- 4217 \* [attestation signature](#), in 5
- 4218 \* [attestation statement](#), in 5.3
- 4219 \* [attestation statement format](#), in 5.3
- 4220 \* [attestation statement format identifier](#), in 7.1
- 4221 \* [attestation type](#), in 5.3
- 4222 \* [Authentication](#), in 3
- 4223 \* [Authentication Assertion](#), in 3
- 4224 \* [authentication extension](#), in 8
- 4225 \* [AuthenticationExtensions](#)
  - 4226 + definition of, in 4.6
  - 4227 + (typedef), in 4.6
- 4228 \* [Authenticator](#), in 3
- 4229 \* [AuthenticatorAssertionResponse](#), in 4.2.2
- 4230 \* [AuthenticatorAttachment](#), in 4.4.4
- 4231 \* [AuthenticatorAttestationResponse](#), in 4.2.1
- 4232 \* [authenticatorCancel](#), in 5.2.3
- 4233 \* [authenticator data](#), in 5.1
- 4234 \* [authenticatorData](#), in 4.2.2
- 4235 \* [authenticator data claimed to have been used for the attestation](#),  
4236 in 5.3.2
- 4237 \* [authenticator data for the attestation](#), in 5.3.2
- 4238 \* [authenticator extension](#), in 8
- 4239 \* [authenticator extension input](#), in 8.3
- 4240 \* [authenticator extension output](#), in 8.5
- 4241 \* [Authenticator extension processing](#), in 8.5
- 4242 \* [authenticatorExtensions](#), in 4.7.1
- 4243 \* [authenticatorGetAssertion](#), in 5.2.2
- 4244 \* [authenticatorMakeCredential](#), in 5.2.1
- 4245 \* [AuthenticatorResponse](#), in 4.2
- 4246 \* [authenticatorSelection](#), in 4.4
- 4247 \* [AuthenticatorSelectionCriteria](#), in 4.4.3
- 4248 \* [AuthenticatorSelectionList](#), in 9.4
- 4249 \* [AuthenticatorTransport](#), in 4.7.4
- 4250 \* [Authorization Gesture](#), in 3
- 4251 \* [Base64url Encoding](#), in 2.1
- 4252 \* [Basic Attestation](#), in 5.3.3
- 4253 \* [Biometric Recognition](#), in 3
- 4254 \* [ble](#), in 4.7.4

4393 + Sometime later, the server deregisters this credential due to  
4394 inactivity.  
4395

## 4396 13. Acknowledgements

4399 We thank the following for their contributions to, and thorough review  
4400 of, this specification: Richard Barnes, Dominic Battr, Domenic  
4401 Denicola, Rahul Ghosh, Brad Hill, Jing Jin, Angelo Liao, Anne van  
4402 Kesteren, Ian Kilpatrick, Giridhar Mandyam, Axel Nennker, Kimberly  
4403 Paulhamus, Adam Powers, Yaron Sheffer, Mike West, Jeffrey Yasskin,  
4404 Boris Zbarsky.  
4405

## 4405 Index

### 4405 Terms defined by this specification

- 4409 \* [AAGUID](#), in 10.4
- 4410 \* [alg](#), in 5.3
- 4411 \* [allowCredentials](#), in 5.5
- 4412 \* [Assertion](#), in 4
- 4413 \* [assertion signature](#), in 6
- 4414 \* [attachment modality](#), in 5.4.5
- 4415 \* [Attestation](#), in 4
- 4416 \* [Attestation Certificate](#), in 4
- 4417 \* [Attestation data](#), in 6.3.1
- 4418 \* [attestation key pair](#), in 4
- 4419 \* [attestationObject](#), in 5.2.1
- 4420 \* [attestation object](#), in 6.3
- 4421 \* [attestation private key](#), in 4
- 4422 \* [attestation public key](#), in 4
- 4423 \* [attestation signature](#), in 6
- 4424 \* [attestation statement](#), in 6.3
- 4425 \* [attestation statement format](#), in 6.3
- 4426 \* [attestation statement format identifier](#), in 8.1
- 4427 \* [attestation type](#), in 6.3
- 4428 \* [Authentication](#), in 4
- 4429 \* [Authentication Assertion](#), in 4
- 4430 \* [authentication extension](#), in 9
- 4431 \* [AuthenticationExtensions](#)
  - 4432 + definition of, in 5.6
  - 4433 + (typedef), in 5.6
- 4434 \* [Authenticator](#), in 4
- 4435 \* [AuthenticatorAssertionResponse](#), in 5.2.2
- 4436 \* [AuthenticatorAttachment](#), in 5.4.5
- 4437 \* [authenticatorAttachment](#), in 5.4.4
- 4438 \* [AuthenticatorAttestationResponse](#), in 5.2.1
- 4439 \* [authenticatorCancel](#), in 6.2.3
- 4440 \* [authenticator data](#), in 6.1
- 4441 \* [authenticatorData](#), in 5.2.2
- 4442 \* [authenticator data claimed to have been used for the attestation](#),  
4443 in 6.3.2
- 4444 \* [authenticator data for the attestation](#), in 6.3.2
- 4445 \* [authenticator extension](#), in 9
- 4446 \* [authenticator extension input](#), in 9.3
- 4447 \* [authenticator extension output](#), in 9.5
- 4448 \* [Authenticator extension processing](#), in 9.5
- 4449 \* [authenticatorExtensions](#), in 5.7.1
- 4450 \* [authenticatorGetAssertion](#), in 6.2.2
- 4451 \* [authenticatorMakeCredential](#), in 6.2.1
- 4452 \* [AuthenticatorResponse](#), in 5.2
- 4453 \* [authenticatorSelection](#), in 5.4
- 4454 \* [AuthenticatorSelectionCriteria](#), in 5.4.4
- 4455 \* [AuthenticatorSelectionList](#), in 10.4
- 4456 \* [AuthenticatorTransport](#), in 5.7.4
- 4457 \* [Authorization Gesture](#), in 4
- 4458 \* [Base64url Encoding](#), in 3
- 4459 \* [Basic Attestation](#), in 6.3.3
- 4460 \* [Biometric Recognition](#), in 4
- 4461 \* [ble](#), in 5.7.4



4255 \* CBOR, in 2.1  
 4256 \* Ceremony, in 3  
 4257 \* challenge  
 4258   + dict-member for MakePublicKeyCredentialOptions, in 4.4  
 4259   + dict-member for PublicKeyCredentialRequestOptions, in 4.5  
 4260   + dict-member for CollectedClientData, in 4.7.1  
 4261 \* Client, in 3  
 4262   client data, in 4.7.1  
 4263   clientDataJSON, in 4.2  
 4264   client extension, in 8  
 4265   client extension input, in 8.3  
 4266   client extension output, in 8.4  
 4267   Client extension processing, in 8.4  
 4268   clientExtensionResults, in 4.1  
 4269   clientExtensions, in 4.7.1  
 4270   Client-Side, in 3  
 4271   client-side credential private key storage, in 3  
 4272   Client-side-resident Credential Private Key, in 3  
 4273   CollectedClientData, in 4.7.1  
 4274   Conforming User Agent, in 3  
  
 4275 \* COSEAlgorithmIdentifier  
 4276   + definition of, in 4.7.5  
 4277   + (typedef), in 4.7.5  
 4278 \* [[Create]](options), in 4.1.3  
 4279   credential key pair, in 3  
 4280   credential private key, in 3  
 4281   Credential Public Key, in 3  
 4282   cross-platform attached, in 4.4.4  
 4283   cross-platform attachment, in 4.4.4  
 4284   DAA, in 5.3.3  
 4285   [[DiscoverFromExternalSource]](options), in 4.1.4  
 4286   [[discovery]], in 4.1  
 4287   displayName, in 4.4.2  
 4288   ECDAA, in 5.3.3  
 4289   ECDAA-Issuer public key, in 7.2  
 4290   Elliptic Curve based Direct Anonymous Attestation, in 5.3.3  
 4291   excludeCredentials, in 4.4  
 4292   extension identifier, in 8.1  
  
 4293 \* extensions  
 4294   + dict-member for MakePublicKeyCredentialOptions, in 4.4  
 4295   + dict-member for PublicKeyCredentialRequestOptions, in 4.5  
 4296   hashAlgorithm, in 4.7.1  
 4297   Hash of the serialized client data, in 4.7.1  
 4298   icon, in 4.4.1  
 4299   id  
 4300   + dict-member for PublicKeyCredentialEntity, in 4.4.1  
 4301   + dict-member for PublicKeyCredentialDescriptor, in 4.7.3  
 4302   [[identifier]], in 4.1  
 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

4462 \* CBOR, in 3  
 4463 \* Ceremony, in 4  
 4464 \* challenge  
 4465   + dict-member for MakePublicKeyCredentialOptions, in 5.4  
 4466   + dict-member for PublicKeyCredentialRequestOptions, in 5.5  
 4467   + dict-member for CollectedClientData, in 5.7.1  
 4468 \* Client, in 4  
 4469   client data, in 5.7.1  
 4470   clientDataJSON, in 5.2  
 4471   client extension, in 9  
 4472   client extension input, in 9.3  
 4473   client extension output, in 9.4  
 4474   Client extension processing, in 9.4  
 4475   clientExtensionResults, in 5.1  
 4476   clientExtensions, in 5.7.1  
 4477   Client-Side, in 4  
 4478   client-side credential private key storage, in 4  
 4479   Client-side-resident Credential Private Key, in 4  
 4480   CollectedClientData, in 5.7.1  
 4481   [[CollectFromCredentialStore]](options), in 5.1.4  
 4482   Conforming User Agent, in 4  
 4483 \* COSEAlgorithmIdentifier  
 4484   + definition of, in 5.7.5  
 4485   + (typedef), in 5.7.5  
 4486 \* [[Create]](options), in 5.1.3  
 4487   credential key pair, in 4  
 4488   credential private key, in 4  
 4489   Credential Public Key, in 4  
 4490   "cross-platform", in 5.4.5  
 4491   cross-platform, in 5.4.5  
 4492   cross-platform attached, in 5.4.5  
 4493   cross-platform attachment, in 5.4.5  
 4494   DAA, in 6.3.3  
 4495   [[DiscoverFromExternalSource]](options), in 5.1.4.1  
 4496   [[discovery]], in 5.1  
 4497   displayName, in 5.4.3  
 4498   ECDAA, in 6.3.3  
 4499   ECDAA-Issuer public key, in 8.2  
 4500   Elliptic Curve based Direct Anonymous Attestation, in 6.3.3  
 4501   excludeCredentials, in 5.4  
 4502   extension identifier, in 9.1  
 4503 \* extensions  
 4504   + dict-member for MakePublicKeyCredentialOptions, in 5.4  
 4505   + dict-member for PublicKeyCredentialRequestOptions, in 5.5  
 4506   hashAlgorithm, in 5.7.1  
 4507   Hash of the serialized client data, in 5.7.1  
 4508   icon, in 5.4.1  
 4509   id  
 4510   + dict-member for PublicKeyCredentialRpEntity, in 5.4.2  
 4511   + dict-member for PublicKeyCredentialUserEntity, in 5.4.3  
 4512   + dict-member for PublicKeyCredentialDescriptor, in 5.7.3  
 4513   [[identifier]], in 5.1  
 4514   identifier of the ECDAA-Issuer public key, in 8.2  
 4515   isPlatformAuthenticatorAvailable(), in 5.1.6  
 4516   JSON-serialized client data, in 5.7.1  
 4517   MakePublicKeyCredentialOptions, in 5.4  
 4518   name, in 5.4.1  
 4519   nfc, in 5.7.4  
 4520   origin, in 5.7.1  
 4521   platform, in 5.4.5  
 4522   "platform", in 5.4.5  
 4523   platform attachment, in 5.4.5  
 4524   platform authenticators, in 5.4.5  
 4525   Privacy CA, in 6.3.3  
 4526   pubKeyCredParams, in 5.4  
 4527 \* publicKey  
 4528   + dict-member for CredentialCreationOptions, in 5.1.1  
 4529   + dict-member for CredentialRequestOptions, in 5.1.2  
 4530   public-key, in 5.7.2  
 4531   Public Key Credential, in 4

4321 \* [PublicKeyCredential](#), in 4.1  
 4322 \* [PublicKeyCredentialDescriptor](#), in 4.7.3  
 4323 \* [PublicKeyCredentialEntity](#), in 4.4.1  
 4324 \* [PublicKeyCredentialParameters](#), in 4.3  
 4325 \* [PublicKeyCredentialRequestOptions](#), in 4.5  
 4326 \* [PublicKeyCredentialType](#), in 4.7.2  
 4327 \* [PublicKeyCredentialUserEntity](#), in 4.4.2  
 4328 \* [Rate Limiting](#), in 3  
 4329 \* [rawId](#), in 4.1  
 4330 \* [Registration](#), in 3  
 4331 \* [registration extension](#), in 8  
 4332 \* [Relying Party](#), in 3  
 4333 \* [Relying Party Identifier](#), in 3  
 4334 \* [response](#), in 4.1  
 4335 \* [rk](#), in 4.4.3  
 4336 \* [roaming authenticators](#), in 4.4.4  
 4337 \* [rp](#), in 4.4  
 4338 \* [rpId](#), in 4.5  
 4339 \* [RP ID](#), in 3  
 4340 \* [Self Attestation](#), in 5.3.3  
 4341 \* [signature](#), in 4.2.2  
 4342 \* [Signing procedure](#), in 5.3.2  
 4343 \* [Test of User Presence](#), in 3

4344 \* [timeout](#)  
 4345     + dict-member for [MakePublicKeyCredentialOptions](#), in 4.4  
 4346     + dict-member for [PublicKeyCredentialRequestOptions](#), in 4.5  
 4347 \* [tokenBindingId](#), in 4.7.1  
 4348 \* [transports](#), in 4.7.3  
 4349 \* [\[\[type\]\]](#), in 4.1  
 4350 \* [type](#)  
 4351     + dict-member for [PublicKeyCredentialParameters](#), in 4.3  
 4352     + dict-member for [PublicKeyCredentialDescriptor](#), in 4.7.3  
 4353 \* [UP](#), in 3  
 4354 \* [usb](#), in 4.7.4  
 4355 \* [user](#), in 4.4  
 4356 \* [User Consent](#), in 3  
 4357 \* [User Present](#), in 3  
 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  
 4368 **Terms defined by reference**  
 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](#)

4532 \* [PublicKeyCredential](#), in 5.1  
 4533 \* [PublicKeyCredentialDescriptor](#), in 5.7.3  
 4534 \* [PublicKeyCredentialEntity](#), in 5.4.1  
 4535 \* [PublicKeyCredentialParameters](#), in 5.3  
 4536 \* [PublicKeyCredentialRequestOptions](#), in 5.5  
 4537 \* [PublicKeyCredentialRpEntity](#), in 5.4.2  
 4538 \* [PublicKeyCredentialType](#), in 5.7.2  
 4539 \* [PublicKeyCredentialUserEntity](#), in 5.4.3  
 4540 \* [Rate Limiting](#), in 4  
 4541 \* [rawId](#), in 5.1  
 4542 \* [Registration](#), in 4  
 4543 \* [registration extension](#), in 9  
 4544 \* [Relying Party](#), in 4  
 4545 \* [Relying Party Identifier](#), in 4  
 4546 \* [requireResidentKey](#), in 5.4.4  
 4547 \* [requireUserVerification](#), in 5.4.4  
 4548 \* [response](#), in 5.1  
 4549 \* [roaming authenticators](#), in 5.4.5  
 4550 \* [rp](#), in 5.4  
 4551 \* [rpId](#), in 5.5  
 4552 \* [RP ID](#), in 4  
 4553 \* [Self Attestation](#), in 6.3.3  
 4554 \* [signature](#), in 5.2.2  
 4555 \* [Signature Counter](#), in 6.1.1  
 4556 \* [Signing procedure](#), in 6.3.2  
 4557 \* [\[\[Store\]\]\(credential\)](#), in 5.1.5  
 4558 \* [Test of User Presence](#), in 4  
 4559 \* [timeout](#)  
 4560     + dict-member for [MakePublicKeyCredentialOptions](#), in 5.4  
 4561     + dict-member for [PublicKeyCredentialRequestOptions](#), in 5.5  
 4562 \* [tokenBindingId](#), in 5.7.1  
 4563 \* [transports](#), in 5.7.3  
 4564 \* [\[\[type\]\]](#), in 5.1  
 4565 \* [type](#)  
 4566     + dict-member for [PublicKeyCredentialParameters](#), in 5.3  
 4567     + dict-member for [PublicKeyCredentialDescriptor](#), in 5.7.3  
 4568 \* [UP](#), in 4  
 4569 \* [usb](#), in 5.7.4  
 4570 \* [user](#), in 5.4  
 4571 \* [User Consent](#), in 4  
 4572 \* [userHandle](#), in 5.2.2  
 4573 \* [User Handle](#), in 4  
 4574 \* [User Present](#), in 4  
 4575 \* [User Verification](#), in 4  
 4576 \* [User Verified](#), in 4  
 4577 \* [UV](#), in 4  
 4578 \* [Verification procedures](#), in 6.3.2  
 4579 \* [Web Authentication API](#), in 5  
 4580 \* [WebAuthn Client](#), in 4

4581  
 4582 **Terms defined by reference**  
 4583  
 4584 \* [\[CREDENTIAL-MANAGEMENT-1\]](#) defines the following terms:  
 4585     + [Credential](#)  
 4586     + [CredentialCreationOptions](#)  
 4587     + [CredentialRequestOptions](#)  
 4588     + [CredentialsContainer](#)  
 4589     + [\[\[CollectFromCredentialStore\]\]\(options\)](#)  
 4590     + [\[\[Store\]\]\(credential\)](#)  
 4591     + [\[\[discovery\]\]](#)  
 4592     + [\[\[type\]\]](#)  
 4593     + [create\(\)](#)  
 4594     + [credential](#)  
 4595     + [credential source](#)  
 4596     + [get\(\)](#)  
 4597     + [id](#)  
 4598     + [remote](#)  
 4599     + [store\(\)](#)  
 4600     + [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  
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

4600 + user mediation  
4601 \* [ECMAScript] defines the following terms:  
4602 + %arraybuffer%  
4603 + internal slot  
4604 + stringify  
4605 \* [ENCODING] defines the following terms:  
4606 + utf-8 encode  
4607 \* [HTML] defines the following terms:  
4608 + ascii serialization of an origin  
4609 + dom manipulation task source  
4610 + effective domain  
4611 + global object  
4612 + in parallel  
4613 + is a registrable domain suffix of or is equal to  
4614 + is not a registrable domain suffix of and is not equal to  
4615 + origin  
4616 + promise  
4617 + relevant settings object  
4618 + task  
4619 + task source  
4620 \* [HTML52] defines the following terms:  
4621 + document.domain  
4622 + opaque origin  
4623 + origin  
4624 \* [INFRA] defines the following terms:  
4625 + append (for list)  
4626 + append (for set)  
4627 + byte sequence  
4628 + continue  
4629 + for each (for list)  
4630 + for each (for map)  
4631 + is empty  
4632 + is not empty  
4633 + item  
4634 + list  
4635 + map  
4636 + ordered set  
4637 + remove  
4638 + set  
4639 \* [mixed-content] defines the following terms:  
4640 + a priori authenticated url  
4641 \* [secure-contexts] defines the following terms:  
4642 + secure context  
4643 \* [TokenBinding] defines the following terms:  
4644 + token binding  
4645 + token binding id  
4646 \* [URL] defines the following terms:  
4647 + domain  
4648 + empty host  
4649 + host  
4650 + ipv4 address  
4651 + ipv6 address  
4652 + opaque host  
4653 + url serializer  
4654 + valid domain  
4655 + valid domain string  
4656 \* [WebCryptoAPI] defines the following terms:  
4657 + recognized algorithm name  
4658 \* [WebIDL] defines the following terms:  
4659 + ArrayBuffer  
4660 + BufferSource  
4661 + ConstraintError  
4662 + DOMException  
4663 + DOMString  
4664 + NotAllowedError  
4665 + NotFoundError  
4666 + NotSupportedError  
4667 + Promise  
4668 + SameObject  
4669 + SecureContext  
4670



4450 + SecurityError  
4451 + TypeError  
4452 + USVString  
4453 + UnknownError  
4454 + Unscopable  
4455 + boolean  
4456 + interface object  
4457 + long  
4458 + present  
4459 + simple exception  
4460 + unsigned long  
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 ECDAAs Algorithm. FIDO Alliance  
4490 Implementation Draft. URL:  
4491 https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-ec-  
4492 daa-algorithm-v1.1-id-20170202.html  
4493  
4494 [FIDOReg]  
4495 R. Lindemann; D. Baghdasaryan; B. Hill. FIDO UAF Registry of  
4496 Predefined Values. FIDO Alliance Proposed Standard. URL:  
4497 https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-ua-  
4498 f-reg-v1.0-ps-20141208.html  
4499  
4500 [HTML]  
4501 Anne van Kesteren; et al. HTML Standard. Living Standard. URL:  
4502 https://html.spec.whatwg.org/multipage/  
4503  
4504 [HTML52]  
4505 Steve Faulkner; et al. HTML 5.2. URL:  
4506 https://www.w3.org/TR/html52/  
4507

4671 + SecurityError  
4672 + TypeError  
4673 + USVString  
4674 + UnknownError  
4675  
4676 + boolean  
4677 + interface object  
4678 + long  
4679 + present  
4680 + simple exception  
4681 + unsigned long  
4682  
4683 References  
4684  
4685 Normative References  
4686  
4687 [CDDL]  
4688 C. Vigano; H. Birkholz. CBOR data definition language (CDDL): a  
4689 notational convention to express CBOR data structures. 21  
4690 September 2016. Internet Draft (work in progress). URL:  
4691 https://tools.ietf.org/html/draft-greevenbosch-appsawg-cbor-cddl  
4692  
4693 [CREDENTIAL-MANAGEMENT-1]  
4694 Mike West. Credential Management Level 1. 4 August 2017. WD.  
4695 URL: https://www.w3.org/TR/credential-management-1/  
4696  
4697 [DOM4]  
4698 Anne van Kesteren. DOM Standard. Living Standard. URL:  
4699 https://dom.spec.whatwg.org/  
4700  
4701 [ECMAScript]  
4702 ECMAScript Language Specification. URL:  
4703 https://tc39.github.io/ecma262/  
4704  
4705 [ENCODING]  
4706 Anne van Kesteren. Encoding Standard. Living Standard. URL:  
4707 https://encoding.spec.whatwg.org/  
4708  
4709 [FIDO-CTAP]  
4710 R. Lindemann; et al. FIDO 2.0: Client to Authenticator Protocol.  
4711 FIDO Alliance Review Draft. URL:  
4712 https://fidoalliance.org/specs/fido-v2.0-rd-20170927/fido-client-  
4713 -to-authenticator-protocol-v2.0-rd-20170927.html  
4714  
4715 [FIDO-U2F-Message-Formats]  
4716 D. Balfanz; J. Ehrensvar; J. Lang. FIDO U2F Raw Message  
4717 Formats. FIDO Alliance Implementation Draft. URL:  
4718 https://fidoalliance.org/specs/fido-u2f-v1.1-id-20160915/fido-u2-  
4719 f-raw-message-formats-v1.1-id-20160915.html  
4720  
4721 [FIDOEcdaaAlgorithm]  
4722 R. Lindemann; et al. FIDO ECDAAs Algorithm. FIDO Alliance  
4723 Implementation Draft. URL:  
4724 https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-ec-  
4725 daa-algorithm-v1.1-id-20170202.html  
4726  
4727 [FIDOReg]  
4728 R. Lindemann; D. Baghdasaryan; B. Hill. FIDO UAF Registry of  
4729 Predefined Values. FIDO Alliance Proposed Standard. URL:  
4730 https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-ua-  
4731 f-reg-v1.0-ps-20141208.html  
4732  
4733 [HTML]  
4734 Anne van Kesteren; et al. HTML Standard. Living Standard. URL:  
4735 https://html.spec.whatwg.org/multipage/  
4736  
4737 [HTML52]  
4738 Steve Faulkner; et al. HTML 5.2. 8 August 2017. CR. URL:  
4739 https://www.w3.org/TR/html52/  
4740

450E [IANA-COSE-ALGS-REG]  
450F IANA CBOR Object Signing and Encryption (COSE) Algorithms  
451C 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  
4522 [RFC4648]  
4523 S. Josefsson. The Base16, Base32, and Base64 Data Encodings.  
4524 October 2006. Proposed Standard. URL:  
4525 <https://tools.ietf.org/html/rfc4648>  
4526  
4527 [RFC5234]  
4528 D. Crocker, Ed.; P. Overell. Augmented BNF for Syntax  
4529 Specifications: ABNF. January 2008. Internet Standard. URL:  
4530 <https://tools.ietf.org/html/rfc5234>  
4531  
4532 [RFC5890]  
4533 J. Klensin. Internationalized Domain Names for Applications  
4534 (IDNA): Definitions and Document Framework. August 2010.  
4535 Proposed Standard. URL: <https://tools.ietf.org/html/rfc5890>  
4536  
4537 [RFC7049]  
4538 C. Bormann; P. Hoffman. Concise Binary Object Representation  
4539 (CBOR). October 2013. Proposed Standard. URL:  
4540 <https://tools.ietf.org/html/rfc7049>  
4541  
4542 [RFC8152]  
4543 J. Schaad. CBOR Object Signing and Encryption (COSE). July 2017.  
4544 Proposed Standard. URL: <https://tools.ietf.org/html/rfc8152>  
4545

4546 [SECURE-CONTEXTS]  
4547 Mike West. Secure Contexts. URL:  
4548 <https://www.w3.org/TR/secure-contexts/>  
4549  
4550 [TokenBinding]  
4551 A. Popov; et al. The Token Binding Protocol Version 1.0.  
4552 February 16, 2017. Internet-Draft. URL:  
4553 <https://tools.ietf.org/html/draft-ietf-tokbind-protocol>  
4554  
4555 [URL]  
4556 Anne van Kesteren. URL Standard. Living Standard. URL:  
4557 <https://url.spec.whatwg.org/>  
4558  
4559 [WebAuthn-Registries]  
4560 Jeff Hodges; Giridhar Mandyam; Michael B. Jones. Registries for  
4561 Web Authentication (WebAuthn). March 2017. Active  
4562 Internet-Draft. URL:  
4563 [https://xml2rfc.tools.ietf.org/cgi-bin/xml2rfc.cgi?modeAsFormat=  
4564 html&ascii&url=https://raw.githubusercontent.com/w3c/webauthn/ma  
4565 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)  
4566  
4567 [WebCryptoAPI]  
4568 Mark Watson. Web Cryptography API. URL:  
4569 <https://www.w3.org/TR/WebCryptoAPI/>

474C [IANA-COSE-ALGS-REG]  
474D IANA CBOR Object Signing and Encryption (COSE) Algorithms  
474E Registry. URL:  
474F <https://www.iana.org/assignments/cose/cose.xhtml#algorithms>  
4750  
4751 [INFRA]  
4752 Anne van Kesteren; Domenic Denicola. Infra Standard. Living  
4753 Standard. URL: <https://infra.spec.whatwg.org/>  
4754  
4755 [MIXED-CONTENT]  
4756 Mike West. Mixed Content. 2 August 2016. CR. URL:  
4757 <https://www.w3.org/TR/mixed-content/>  
4758  
4759 [RFC2119]  
4760 S. Bradner. Key words for use in RFCs to Indicate Requirement  
4761 Levels. March 1997. Best Current Practice. URL:  
4762 <https://tools.ietf.org/html/rfc2119>  
4763  
4764 [RFC4648]  
4765 S. Josefsson. The Base16, Base32, and Base64 Data Encodings.  
4766 October 2006. Proposed Standard. URL:  
4767 <https://tools.ietf.org/html/rfc4648>  
4768  
4769 [RFC5234]  
4770 D. Crocker, Ed.; P. Overell. Augmented BNF for Syntax  
4771 Specifications: ABNF. January 2008. Internet Standard. URL:  
4772 <https://tools.ietf.org/html/rfc5234>  
4773  
4774 [RFC5890]  
4775 J. Klensin. Internationalized Domain Names for Applications  
4776 (IDNA): Definitions and Document Framework. August 2010.  
4777 Proposed Standard. URL: <https://tools.ietf.org/html/rfc5890>  
4778  
4779 [RFC7049]  
4780 C. Bormann; P. Hoffman. Concise Binary Object Representation  
4781 (CBOR). October 2013. Proposed Standard. URL:  
4782 <https://tools.ietf.org/html/rfc7049>  
4783  
4784 [RFC8152]  
4785 J. Schaad. CBOR Object Signing and Encryption (COSE). July 2017.  
4786 Proposed Standard. URL: <https://tools.ietf.org/html/rfc8152>  
4787  
4788 [SEC1]  
4789 SEC1: Elliptic Curve Cryptography, Version 2.0. URL:  
4790 <http://www.secg.org/sec1-v2.pdf>  
4791  
4792 [SECURE-CONTEXTS]  
4793 Mike West. Secure Contexts. 15 September 2016. CR. URL:  
4794 <https://www.w3.org/TR/secure-contexts/>  
4795  
4796 [TokenBinding]  
4797 A. Popov; et al. The Token Binding Protocol Version 1.0.  
4798 February 16, 2017. Internet-Draft. URL:  
4799 <https://tools.ietf.org/html/draft-ietf-tokbind-protocol>  
4800  
4801 [URL]  
4802 Anne van Kesteren. URL Standard. Living Standard. URL:  
4803 <https://url.spec.whatwg.org/>  
4804  
4805 [WebAuthn-Registries]  
4806 Jeff Hodges; Giridhar Mandyam; Michael B. Jones. Registries for  
4807 Web Authentication (WebAuthn). March 2017. Active  
4808 Internet-Draft. URL:  
4809 [https://xml2rfc.tools.ietf.org/cgi-bin/xml2rfc.cgi?modeAsFormat=  
4810 html&ascii&url=https://raw.githubusercontent.com/w3c/webauthn/ma  
4811 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)  
4812  
4813 [WebCryptoAPI]  
4814 Mark Watson. Web Cryptography API. 26 January 2017. REC. URL:  
4815 <https://www.w3.org/TR/WebCryptoAPI/>

4570 [WebIDL]  
 4571 Cameron McCormack; Boris Zbarsky; Tobie Langel. Web IDL. URL:  
 4572 <https://heycam.github.io/webidl/>  
 4573  
 4574 [WebIDL-1]  
 4575 Cameron McCormack. WebIDL Level 1. URL:  
 4576 <https://www.w3.org/TR/2016/REC-WebIDL-1-20161215/>  
 4577  
 4578 Informative References  
 4579 [Ceremony]  
 4580 Carl Ellison. Ceremony Design and Analysis. 2007. URL:  
 4581 <https://eprint.iacr.org/2007/399.pdf>  
 4582  
 4583 [FIDO-APPID]  
 4584 D. Balfanz; et al. FIDO AppID and Facets. FIDO Alliance Review  
 4585 Draft. URL:  
 4586 <https://fidoalliance.org/specs/fido-uaf-v1.1-rd-20161005/fido-ap>  
 4587 [pid-and-facets-v1.1-rd-20161005.html](https://fidoalliance.org/specs/fido-uaf-v1.1-rd-20161005/fido-ap)  
 4588  
 4589 [FIDO-U2F-Message-Formats]  
 4590 D. Balfanz; J. Ehrensvar; J. Lang. FIDO U2F Raw Message  
 4591 Formats. FIDO Alliance Implementation Draft. URL:  
 4592 <https://fidoalliance.org/specs/fido-u2f-v1.1-id-20160915/fido-u2>  
 4593 [f-raw-message-formats-v1.1-id-20160915.html](https://fidoalliance.org/specs/fido-u2f-v1.1-id-20160915/fido-u2)  
 4594  
 4595 [FIDOMetadataService]  
 4596 R. Lindemann; B. Hill; D. Baghdasaryan. FIDO Metadata Service  
 4597 v1.0. FIDO Alliance Proposed Standard. URL:  
 4598 <https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-ua>  
 4600 [f-metadata-service-v1.0-ps-20141208.html](https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-ua)  
 4601  
 4602 [FIDOSecRef]  
 4603 R. Lindemann; D. Baghdasaryan; B. Hill. FIDO Security Reference.  
 4604 FIDO Alliance Proposed Standard. URL:  
 4605 <https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-se>  
 4606 [curity-ref-v1.0-ps-20141208.html](https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-se)  
 4607  
 4608 [GeoJSON]  
 4609 The GeoJSON Format Specification. URL:  
 4610 <http://geojson.org/geojson-spec.html>  
 4611  
 4612 [ISOBiometricVocabulary]  
 4613 ISO/IEC JTC1/SC37. Information technology -- Vocabulary --  
 4614 Biometrics. 15 December 2012. International Standard: ISO/IEC  
 4615 2382-37:2012(E) First Edition. URL:  
 4616 <http://standards.iso.org/ittf/PubliclyAvailableStandards/c055194>  
 4617 [\\_ISOIEC\\_2382-37\\_2012.zip](http://standards.iso.org/ittf/PubliclyAvailableStandards/c055194)  
 4618  
 4619 [RFC4949]  
 4620 R. Shirey. Internet Security Glossary, Version 2. August 2007.  
 4621 Informational. URL: <https://tools.ietf.org/html/rfc4949>  
 4622  
 4623 [RFC5280]  
 4624 D. Cooper; et al. Internet X.509 Public Key Infrastructure  
 4625 Certificate and Certificate Revocation List (CRL) Profile. May  
 4626 2008. Proposed Standard. URL:  
 4627 <https://tools.ietf.org/html/rfc5280>  
 4628  
 4629 [RFC6265]  
 4630 A. Barth. HTTP State Management Mechanism. April 2011. Proposed  
 4631 Standard. URL: <https://tools.ietf.org/html/rfc6265>  
 4632  
 4633 [RFC6454]  
 4634 A. Barth. The Web Origin Concept. December 2011. Proposed  
 4635 Standard. URL: <https://tools.ietf.org/html/rfc6454>  
 4636  
 4637 [RFC7515]  
 4638 M. Jones; J. Bradley; N. Sakimura. JSON Web Signature (JWS). May  
 4639

4810 [WebIDL]  
 4811 Cameron McCormack; Boris Zbarsky; Tobie Langel. Web IDL. 15  
 4812 December 2016. ED. URL: <https://heycam.github.io/webidl/>  
 4813  
 4814 [WebIDL-1]  
 4815 Cameron McCormack. WebIDL Level 1. 15 December 2016. REC. URL:  
 4816 <https://www.w3.org/TR/2016/REC-WebIDL-1-20161215/>  
 4817  
 4818 Informative References  
 4819 [Ceremony]  
 4820 Carl Ellison. Ceremony Design and Analysis. 2007. URL:  
 4821 <https://eprint.iacr.org/2007/399.pdf>  
 4822  
 4823 [FIDO-APPID]  
 4824 D. Balfanz; et al. FIDO AppID and Facets. FIDO Alliance Review  
 4825 Draft. URL:  
 4826 <https://fidoalliance.org/specs/fido-uaf-v1.1-rd-20161005/fido-ap>  
 4827 [pid-and-facets-v1.1-rd-20161005.html](https://fidoalliance.org/specs/fido-uaf-v1.1-rd-20161005/fido-ap)  
 4828  
 4829 [FIDOMetadataService]  
 4830 R. Lindemann; B. Hill; D. Baghdasaryan. FIDO Metadata Service  
 4831 v1.0. FIDO Alliance Proposed Standard. URL:  
 4832 <https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-ua>  
 4833 [f-metadata-service-v1.0-ps-20141208.html](https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-ua)  
 4834  
 4835 [FIDOSecRef]  
 4836 R. Lindemann; D. Baghdasaryan; B. Hill. FIDO Security Reference.  
 4837 FIDO Alliance Proposed Standard. URL:  
 4838 <https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-se>  
 4839 [curity-ref-v1.0-ps-20141208.html](https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-se)  
 4840  
 4841 [GeoJSON]  
 4842 The GeoJSON Format Specification. URL:  
 4843 <http://geojson.org/geojson-spec.html>  
 4844  
 4845 [ISOBiometricVocabulary]  
 4846 ISO/IEC JTC1/SC37. Information technology -- Vocabulary --  
 4847 Biometrics. 15 December 2012. International Standard: ISO/IEC  
 4848 2382-37:2012(E) First Edition. URL:  
 4849 <http://standards.iso.org/ittf/PubliclyAvailableStandards/c055194>  
 4850 [\\_ISOIEC\\_2382-37\\_2012.zip](http://standards.iso.org/ittf/PubliclyAvailableStandards/c055194)  
 4851  
 4852 [RFC4949]  
 4853 R. Shirey. Internet Security Glossary, Version 2. August 2007.  
 4854 Informational. URL: <https://tools.ietf.org/html/rfc4949>  
 4855  
 4856 [RFC5280]  
 4857 D. Cooper; et al. Internet X.509 Public Key Infrastructure  
 4858 Certificate and Certificate Revocation List (CRL) Profile. May  
 4859 2008. Proposed Standard. URL:  
 4860 <https://tools.ietf.org/html/rfc5280>  
 4861  
 4862 [RFC6265]  
 4863 A. Barth. HTTP State Management Mechanism. April 2011. Proposed  
 4864 Standard. URL: <https://tools.ietf.org/html/rfc6265>  
 4865  
 4866 [RFC6454]  
 4867 A. Barth. The Web Origin Concept. December 2011. Proposed  
 4868 Standard. URL: <https://tools.ietf.org/html/rfc6454>  
 4869  
 4870 [RFC7515]  
 4871 M. Jones; J. Bradley; N. Sakimura. JSON Web Signature (JWS). May  
 4872



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  
 4658 [TPMv2-Part2]  
 4659 Trusted Platform Module Library, Part 2: Structures. URL:  
 4660 http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-  
 4661 2.0-Part-2-Structures-01.38.pdf  
 4662  
 4663 [TPMv2-Part3]  
 4664 Trusted Platform Module Library, Part 3: Commands. URL:  
 4665 http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-  
 4666 2.0-Part-3-Commands-01.38.pdf  
 4667  
 4668 [UAFProtocol]  
 4669 R. Lindemann; et al. FIDO UAF Protocol Specification v1.0. FIDO  
 4670 Alliance Proposed Standard. URL:  
 4671 https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-ua  
 4672 f-protocol-v1.0-ps-20141208.html  
 4673  
 4674 IDL Index  
 4675  
 4676 [SecureContext]  
 4677 interface PublicKeyCredential : Credential {  
 4678 [SameObject] readonly attribute ArrayBuffer rawId;  
 4679 [SameObject] readonly attribute AuthenticatorResponse response;  
 4680 [SameObject] readonly attribute AuthenticationExtensions clientExtensionResu  
 4681 lts;  
 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;

4874 2015. Proposed Standard. URL:  
 4875 https://tools.ietf.org/html/rfc7515  
 4876  
 4877 [RFC8017]  
 4878 K. Moriarty, Ed.; et al. PKCS #1: RSA Cryptography  
 4879 Specifications Version 2.2. November 2016. Informational. URL:  
 4880 https://tools.ietf.org/html/rfc8017  
 4881  
 4882 [TPMv2-EK-Profile]  
 4883 TCG EK Credential Profile for TPM Family 2.0. URL:  
 4884 http://www.trustedcomputinggroup.org/wp-content/uploads/Credenti  
 4885 al\_Profile\_EK\_V2.0\_R14\_published.pdf  
 4886  
 4887 [TPMv2-Part1]  
 4888 Trusted Platform Module Library, Part 1: Architecture. URL:  
 4889 http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-  
 4890 2.0-Part-1-Architecture-01.38.pdf  
 4891  
 4892 [TPMv2-Part2]  
 4893 Trusted Platform Module Library, Part 2: Structures. URL:  
 4894 http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-  
 4895 2.0-Part-2-Structures-01.38.pdf  
 4896  
 4897 [TPMv2-Part3]  
 4898 Trusted Platform Module Library, Part 3: Commands. URL:  
 4899 http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-  
 4900 2.0-Part-3-Commands-01.38.pdf  
 4901  
 4902 [UAFProtocol]  
 4903 R. Lindemann; et al. FIDO UAF Protocol Specification v1.0. FIDO  
 4904 Alliance Proposed Standard. URL:  
 4905 https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-ua  
 4906 f-protocol-v1.0-ps-20141208.html  
 4907  
 4908 IDL Index  
 4909  
 4910 [SecureContext]  
 4911 interface PublicKeyCredential : Credential {  
 4912 [SameObject] readonly attribute ArrayBuffer rawId;  
 4913 [SameObject] readonly attribute AuthenticatorResponse response;  
 4914 [SameObject] readonly attribute AuthenticationExtensions clientExtensionResu  
 4915 lts;  
 4916 };  
 4917  
 4918 partial dictionary CredentialCreationOptions {  
 4919 MakePublicKeyCredentialOptions publicKey;  
 4920 };  
 4921  
 4922 partial dictionary CredentialRequestOptions {  
 4923 PublicKeyCredentialRequestOptions publicKey;  
 4924 };  
 4925  
 4926 [SecureContext]  
 4927 partial interface PublicKeyCredential {  
 4928 static Promise < boolean > isPlatformAuthenticatorAvailable();  
 4929 };  
 4930  
 4931 [SecureContext]  
 4932 interface AuthenticatorResponse {  
 4933 [SameObject] readonly attribute ArrayBuffer clientDataJSON;  
 4934 };  
 4935  
 4936 [SecureContext]  
 4937 interface AuthenticatorAttestationResponse : AuthenticatorResponse {  
 4938 [SameObject] readonly attribute ArrayBuffer attestationObject;  
 4939 };  
 4940  
 4941 [SecureContext]  
 4942 interface AuthenticatorAssertionResponse : AuthenticatorResponse {  
 4943 [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;
4724
4725     unsigned long timeout;
4726     sequence<PublicKeyCredentialDescriptor> excludeCredentials = [];
4727     AuthenticatorSelectionCriteria authenticatorSelection;
4728     AuthenticationExtensions extensions;
4729 };
4730
4731 dictionary PublicKeyCredentialEntity {
4732     DOMString id;
4733     DOMString name;
4734     USVString icon;
4735 };
4736
4737 dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
4738     DOMString displayName;
4739 };
4740
4741 dictionary AuthenticatorSelectionCriteria {
4742     AuthenticatorAttachment aa; // authenticatorAttachment
4743     boolean rk = false; // requireResidentKey
4744     boolean uv = false; // requireUserVerification
4745 };
4746
4747 enum AuthenticatorAttachment {
4748     "plat" // Platform attachment
4749     "xplat" // Cross-platform attachment
4750 };
4751
4752 dictionary PublicKeyCredentialRequestOptions {
4753     required BufferSource challenge;
4754     unsigned long timeout;
4755     USVString rpId;
4756     sequence<PublicKeyCredentialDescriptor> allowCredentials = [];
4757     AuthenticationExtensions extensions;
4758 };
4759
4760 typedef record<DOMString, any> AuthenticationExtensions;
4761
4762 dictionary CollectedClientData {
4763     required DOMString challenge;
4764     required DOMString origin;
4765     required DOMString hashAlgorithm;
4766     DOMString tokenBindingId;
4767     AuthenticationExtensions clientExtensions;
4768     AuthenticationExtensions authenticatorExtensions;
4769 };
4770
4771 enum PublicKeyCredentialType {
4772     "public-key"
4773 };

```

```

4944 [SameObject] readonly attribute ArrayBuffer signature;
4945 [SameObject] readonly attribute ArrayBuffer userHandle;
4946 };
4947
4948 dictionary PublicKeyCredentialParameters {
4949     required PublicKeyCredentialType type;
4950     required COSEAlgorithmIdentifier alg;
4951 };
4952
4953 dictionary MakePublicKeyCredentialOptions {
4954     required PublicKeyCredentialRpEntity rp;
4955     required PublicKeyCredentialUserEntity user;
4956
4957     required BufferSource challenge;
4958     required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
4959
4960     unsigned long timeout;
4961     sequence<PublicKeyCredentialDescriptor> excludeCredentials = [];
4962     AuthenticatorSelectionCriteria authenticatorSelection;
4963     AuthenticationExtensions extensions;
4964 };
4965
4966 dictionary PublicKeyCredentialEntity {
4967     DOMString name;
4968     USVString icon;
4969 };
4970
4971 dictionary PublicKeyCredentialRpEntity : PublicKeyCredentialEntity {
4972     DOMString id;
4973 };
4974
4975 dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
4976     BufferSource id;
4977     DOMString displayName;
4978 };
4979
4980 dictionary AuthenticatorSelectionCriteria {
4981     AuthenticatorAttachment authenticatorAttachment;
4982     boolean requireResidentKey = false;
4983     boolean requireUserVerification = false;
4984 };
4985
4986 enum AuthenticatorAttachment {
4987     "platform" // Platform attachment
4988     "cross-platform" // Cross-platform attachment
4989 };
4990
4991 dictionary PublicKeyCredentialRequestOptions {
4992     required BufferSource challenge;
4993     unsigned long timeout;
4994     USVString rpId;
4995     sequence<PublicKeyCredentialDescriptor> allowCredentials = [];
4996     AuthenticationExtensions extensions;
4997 };
4998
4999 typedef record<DOMString, any> AuthenticationExtensions;
5000
5001 dictionary CollectedClientData {
5002     required DOMString challenge;
5003     required DOMString origin;
5004     required DOMString hashAlgorithm;
5005     DOMString tokenBindingId;
5006     AuthenticationExtensions clientExtensions;
5007     AuthenticationExtensions authenticatorExtensions;
5008 };
5009
5010 enum PublicKeyCredentialType {
5011     "public-key"
5012 };

```

```

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 typedef long COSEAlgorithmIdentifier;
4787
4788 typedef sequence<AAGUID> AuthenticatorSelectionList;
4789
4790 typedef BufferSource AAGUID;
4791
4792 #base64url-encodingReferenced in:
4793 * 4.1. PublicKeyCredential Interface
4794 * 4.1.3. Create a new credential - PublicKeyCredential's
4795 [[Create]](options) method (2)
4796 * 4.1.4. Use an existing credential to make an assertion -
4797 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
4798 method (2)
4799 * 6.2. Verifying an authentication assertion
4800
4801 #cborReferenced in:
4802 * 4.1.3. Create a new credential - PublicKeyCredential's
4803 [[Create]](options) method
4804 * 4.1.4. Use an existing credential to make an assertion -
4805 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
4806 method
4807 * 5.1. Authenticator data (2)
4808 * 8. WebAuthn Extensions (2) (3)
4809 * 8.2. Defining extensions (2)
4810 * 8.3. Extending request parameters
4811 * 8.4. Client extension processing (2)
4812 * 8.5. Authenticator extension processing (2) (3) (4) (5)
4813
4814 #attestationReferenced in:
4815 * 3. Terminology
4816 * 5. WebAuthn Authenticator model (2)
4817 * 5.3. Attestation (2) (3) (4)
4818
4819 #attestation-certificateReferenced in:
4820 * 3. Terminology (2)
4821 * 7.3.1. TPM attestation statement certificate requirements
4822
4823 #attestation-key-pairReferenced in:
4824 * 3. Terminology (2)
4825 * 5.3. Attestation
4826
4827 #attestation-private-keyReferenced in:
4828 * 5. WebAuthn Authenticator model
4829 * 5.3. Attestation
4830
4831 #attestation-public-keyReferenced in:
4832 * 5.3. Attestation
4833
4834 #authenticationReferenced in:
4835 * 1. Introduction (2)
4836 * 3. Terminology (2) (3) (4) (5) (6) (7)
4837 * 6.2. Verifying an authentication assertion
4838
4839 #authentication-assertionReferenced in:
4840 * 1. Introduction
4841 * 3. Terminology (2) (3)

```

```

5013 dictionary PublicKeyCredentialDescriptor {
5014     required PublicKeyCredentialType type;
5015     required BufferSource id;
5016     sequence<AuthenticatorTransport> transports;
5017 };
5018
5019 enum AuthenticatorTransport {
5020     "usb",
5021     "nfc",
5022     "ble",
5023 };
5024
5025 typedef long COSEAlgorithmIdentifier;
5026
5027 typedef sequence<AAGUID> AuthenticatorSelectionList;
5028
5029 typedef BufferSource AAGUID;
5030
5031 #base64url-encodingReferenced in:
5032 * 5.1. PublicKeyCredential Interface
5033 * 5.1.3. Create a new credential - PublicKeyCredential's
5034 [[Create]](options) method (2)
5035 * 5.1.4.1. PublicKeyCredential's
5036 [[DiscoverFromExternalSource]](options) method (2)
5037 * 7.2. Verifying an authentication assertion
5038
5039 #cborReferenced in:
5040 * 5.1.3. Create a new credential - PublicKeyCredential's
5041 [[Create]](options) method
5042 * 5.1.4.1. PublicKeyCredential's
5043 [[DiscoverFromExternalSource]](options) method
5044 * 6.1. Authenticator data (2)
5045 * 9. WebAuthn Extensions (2) (3)
5046 * 9.2. Defining extensions (2)
5047 * 9.3. Extending request parameters
5048 * 9.4. Client extension processing (2)
5049 * 9.5. Authenticator extension processing (2) (3) (4) (5)
5050
5051 #attestationReferenced in:
5052 * 4. Terminology
5053 * 6. WebAuthn Authenticator model (2)
5054 * 6.3. Attestation (2) (3) (4)
5055
5056 #attestation-certificateReferenced in:
5057 * 4. Terminology (2)
5058 * 8.3.1. TPM attestation statement certificate requirements
5059
5060 #attestation-key-pairReferenced in:
5061 * 4. Terminology (2)
5062 * 6.3. Attestation
5063
5064 #attestation-private-keyReferenced in:
5065 * 6. WebAuthn Authenticator model
5066 * 6.3. Attestation
5067
5068 #attestation-public-keyReferenced in:
5069 * 6.3. Attestation
5070
5071 #authenticationReferenced in:
5072 * 1. Introduction (2)
5073 * 4. Terminology (2) (3) (4) (5) (6) (7)
5074 * 7.2. Verifying an authentication assertion (2) (3)
5075
5076 #authentication-assertionReferenced in:
5077 * 1. Introduction
5078 * 4. Terminology (2) (3)

```



4844 \* 4.1. PublicKeyCredential Interface  
4845 \* 4.2.2. Web Authentication Assertion (interface  
4846 AuthenticatorAssertionResponse)  
4847 \* 4.5. Options for Assertion Generation (dictionary  
4848 PublicKeyCredentialRequestOptions)  
4849 \* 8. WebAuthn Extensions  
4850  
4851 #authenticatorReferenced in:  
4852 \* 1. Introduction (2) (3) (4)  
4853 \* 1.1. Use Cases  
4854 \* 2. Conformance  
4855 \* 3. Terminology (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13)  
4856 (14) (15)  
4857 \* 4. Web Authentication API (2) (3)  
4858 \* 4.1. PublicKeyCredential Interface  
4859 \* 4.1.3. Create a new credential - PublicKeyCredential's  
4860 [[Create]](options) method (2)  
4861 \* 4.1.4. Use an existing credential to make an assertion -  
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

5081 \* 5.1. PublicKeyCredential Interface  
5082 \* 5.2.2. Web Authentication Assertion (interface  
5083 AuthenticatorAssertionResponse)  
5084 \* 5.5. Options for Assertion Generation (dictionary  
5085 PublicKeyCredentialRequestOptions)  
5086 \* 9. WebAuthn Extensions  
5087  
5088 #authenticatorReferenced in:  
5089 \* 1. Introduction (2) (3) (4)  
5090 \* 1.1. Use Cases  
5091 \* 2.2. Authenticators  
5092 \* 4. Terminology (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13)  
5093 (14) (15)  
5094 \* 5. Web Authentication API (2) (3)  
5095 \* 5.1. PublicKeyCredential Interface  
5096 \* 5.1.3. Create a new credential - PublicKeyCredential's  
5097 [[Create]](options) method (2)  
5098 \* 5.1.4.1. PublicKeyCredential's  
5099 [[DiscoverFromExternalSource]](options) method (2) (3)  
5100 \* 5.2. Authenticator Responses (interface AuthenticatorResponse)  
5101 \* 5.2.1. Information about Public Key Credential (interface  
5102 AuthenticatorAttestationResponse) (2)  
5103 \* 5.2.2. Web Authentication Assertion (interface  
5104 AuthenticatorAssertionResponse)  
5105 \* 5.4.5. Authenticator Attachment enumeration (enum  
5106 AuthenticatorAttachment)  
5107 \* 5.5. Options for Assertion Generation (dictionary  
5108 PublicKeyCredentialRequestOptions)  
5109 \* 6. WebAuthn Authenticator model (2) (3) (4) (5) (6)  
5110 \* 6.1. Authenticator data  
5111 \* 6.2.1. The authenticatorMakeCredential operation (2)  
5112 \* 6.2.2. The authenticatorGetAssertion operation (2) (3) (4)  
5113 \* 6.3. Attestation (2) (3) (4) (5) (6) (7) (8) (9)  
5114 \* 6.3.2. Attestation Statement Formats  
5115 \* 6.3.4. Generating an Attestation Object  
5116 \* 6.3.5.1. Privacy  
5117 \* 6.3.5.2. Attestation Certificate and Attestation Certificate CA  
5118 Compromise  
5119 \* 7.1. Registering a new credential  
5120 \* 8.2. Packed Attestation Statement Format  
5121 \* 8.4. Android Key Attestation Statement Format  
5122 \* 8.5. Android SafetyNet Attestation Statement Format  
5123 \* 10.5. Supported Extensions Extension (exts)  
5124 \* 10.6. User Verification Index Extension (uvi)  
5125 \* 10.7. Location Extension (loc) (2) (3) (4)  
5126 \* 10.8. User Verification Method Extension (uvm)  
5127 \* 12. Sample scenarios  
5128  
5129 #authorization-gestureReferenced in:  
5130 \* 1.1.1. Registration  
5131 \* 1.1.2. Authentication  
5132 \* 1.1.3. Other use cases and configurations  
5133 \* 4. Terminology (2) (3) (4) (5) (6)  
5134 \* 5.1.4. Use an existing credential to make an assertion (2)  
5135  
5136 #biometric-recognitionReferenced in:  
5137 \* 4. Terminology (2)  
5138  
5139 #ceremonyReferenced in:  
5140 \* 1. Introduction  
5141 \* 4. Terminology (2) (3) (4) (5) (6) (7)  
5142 \* 7.1. Registering a new credential  
5143 \* 7.2. Verifying an authentication assertion  
5144  
5145 #clientReferenced in:  
5146 \* 4. Terminology  
5147 \* 5.1.6. Platform Authenticator Availability - PublicKeyCredential's  
5148 isPlatformAuthenticatorAvailable() method (2) (3) (4)  
5149

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:  
 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)

5150 #client-side-resident-credential-private-keyReferenced in:  
 5151 \* 4. Terminology (2)  
 5152 \* 5.1.3. Create a new credential - PublicKeyCredential's  
 5153 [[Create]](options) method  
 5154 \* 5.4.4. Authenticator Selection Criteria (dictionary  
 5155 AuthenticatorSelectionCriteria) (2)  
 5156 \* 6.2.1. The authenticatorMakeCredential operation  
 5157  
 5158 #conforming-user-agentReferenced in:  
 5159 \* 1. Introduction  
 5160 \* 2.1. User Agents  
 5161 \* 2.2. Authenticators  
 5162 \* 4. Terminology (2)  
 5163  
 5164 #credential-public-keyReferenced in:  
 5165 \* 4. Terminology (2) (3)  
 5166 \* 5.2.1. Information about Public Key Credential (interface  
 5167 AuthenticatorAttestationResponse)  
 5168 \* 6. WebAuthn Authenticator model  
 5169 \* 6.1. Authenticator data  
 5170 \* 6.3. Attestation (2) (3)  
 5171 \* 6.3.1. Attestation data (2)  
 5172 \* 8.4. Android Key Attestation Statement Format  
 5173 \* 12.1. Registration (2)  
 5174  
 5175 #credential-key-pairReferenced in:  
 5176 \* 4. Terminology (2) (3)  
 5177 \* 5.1.3. Create a new credential - PublicKeyCredential's  
 5178 [[Create]](options) method  
 5179  
 5180 #credential-private-keyReferenced in:  
 5181 \* 4. Terminology (2) (3) (4)  
 5182 \* 5.1. PublicKeyCredential Interface  
 5183 \* 5.2.2. Web Authentication Assertion (interface  
 5184 AuthenticatorAssertionResponse)  
 5185 \* 6. WebAuthn Authenticator model  
 5186 \* 6.2.2. The authenticatorGetAssertion operation  
 5187 \* 6.3. Attestation (2)  
 5188 \* 7.2. Verifying an authentication assertion  
 5189  
 5190 #registrationReferenced in:  
 5191 \* 1. Introduction (2)  
 5192 \* 4. Terminology (2) (3) (4) (5) (6) (7) (8) (9)  
 5193 \* 7.1. Registering a new credential  
 5194  
 5195 #relying-partyReferenced in:  
 5196 \* 1. Introduction (2) (3) (4) (5) (6) (7)  
 5197 \* 1.1.3. Other use cases and configurations  
 5198 \* 2.3. Relying Parties  
 5199 \* 4. Terminology (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13)  
 5200 (14) (15) (16) (17) (18) (19) (20) (21) (22) (23) (24) (25) (26)  
 5201 \* 5. Web Authentication API (2) (3) (4) (5) (6) (7)  
 5202 \* 5.1.4. Use an existing credential to make an assertion  
 5203 \* 5.1.4.1. PublicKeyCredential's  
 5204 [[DiscoverFromExternalSource]](options) method (2)  
 5205 \* 5.1.6. Platform Authenticator Availability - PublicKeyCredential's  
 5206 isPlatformAuthenticatorAvailable() method (2) (3)  
 5207 \* 5.2. Authenticator Responses (interface AuthenticatorResponse)  
 5208 \* 5.2.1. Information about Public Key Credential (interface  
 5209 AuthenticatorAttestationResponse) (2)  
 5210 \* 5.2.2. Web Authentication Assertion (interface  
 5211 AuthenticatorAssertionResponse)  
 5212 \* 5.4. Options for Credential Creation (dictionary  
 5213 MakePublicKeyCredentialOptions) (2) (3) (4) (5) (6)  
 5214 \* 5.4.1. Public Key Entity Description (dictionary  
 5215 PublicKeyCredentialEntity) (2) (3)  
 5216 \* 5.4.2. RP Parameters for Credential Generation (dictionary  
 5217 PublicKeyCredentialRpEntity) (2)  
 5218 \* 5.4.4. Authenticator Selection Criteria (dictionary  
 5219 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  
 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

5220 \* 5.4.5. Authenticator Attachment enumeration (enum  
 5221 AuthenticatorAttachment) (2) (3) (4)  
 5222 \* 5.7.1. Client data used in WebAuthn signatures (dictionary  
 5223 CollectedClientData) (2) (3) (4)  
 5224 \* 5.7.4. Authenticator Transport enumeration (enum  
 5225 AuthenticatorTransport) (2)  
 5226 \* 6. WebAuthn Authenticator model (2)  
 5227 \* 6.1. Authenticator data (2)  
 5228 \* 6.1.1. Signature Counter Considerations (2) (3) (4) (5) (6)  
 5229 \* 6.2.1. The authenticatorMakeCredential operation (2) (3) (4) (5)  
 5230 \* 6.2.2. The authenticatorGetAssertion operation (2) (3)  
 5231 \* 6.3. Attestation (2) (3) (4) (5) (6)  
 5232 \* 6.3.5.1. Privacy  
 5233 \* 6.3.5.2. Attestation Certificate and Attestation Certificate CA  
 5234 Compromise (2) (3) (4) (5) (6)  
 5235 \* 7. Relying Party Operations (2) (3) (4)  
 5236 \* 7.1. Registering a new credential (2) (3) (4) (5) (6) (7) (8) (9)  
 5237 (10) (11) (12)  
 5238 \* 7.2. Verifying an authentication assertion (2) (3) (4) (5) (6) (7)  
 5239 (8)  
 5240 \* 8.4. Android Key Attestation Statement Format  
 5241 \* 9. WebAuthn Extensions (2) (3) (4)  
 5242 \* 9.2. Defining extensions (2)  
 5243 \* 9.3. Extending request parameters (2) (3) (4)  
 5244 \* 9.6. Example Extension (2) (3)  
 5245 \* 10.1. FIDO Appid Extension (appid) (2)  
 5246 \* 10.2. Simple Transaction Authorization Extension (txAuthSimple)  
 5247 \* 10.4. Authenticator Selection Extension (authnSel) (2) (3)  
 5248 \* 10.5. Supported Extensions Extension (exts) (2)  
 5249 \* 10.6. User Verification Index Extension (uvi)  
 5250 \* 10.7. Location Extension (loc) (2)  
 5251 \* 11.2. WebAuthn Extension Identifier Registrations (2)  
 5252 \* 12.1. Registration (2) (3) (4) (5)  
 5253 \* 12.2. Registration Specifically with Platform Authenticator (2) (3)  
 5254 \* 12.3. Authentication (2) (3) (4) (5)  
 5255 \* 12.4. Decommissioning (2)  
 5256  
 5257 #relying-party-identifierReferenced in:  
 5258 \* 5. Web Authentication API  
 5259 \* 5.4. Options for Credential Creation (dictionary  
 5260 MakePublicKeyCredentialOptions)  
 5261 \* 5.5. Options for Assertion Generation (dictionary  
 5262 PublicKeyCredentialRequestOptions)  
 5263 \* 6. WebAuthn Authenticator model  
 5264  
 5265 #rp-idReferenced in:  
 5266 \* 4. Terminology (2) (3) (4) (5) (6)  
 5267 \* 5. Web Authentication API (2) (3) (4) (5)  
 5268 \* 5.1.3. Create a new credential - PublicKeyCredential's  
 5269 [[Create]](options) method (2)  
 5270 \* 5.1.4.1. PublicKeyCredential's  
 5271 [[DiscoverFromExternalSource]](options) method (2)  
 5272 \* 5.4.2. RP Parameters for Credential Generation (dictionary  
 5273 PublicKeyCredentialRpEntity)  
 5274 \* 6. WebAuthn Authenticator model  
 5275 \* 6.1. Authenticator data (2) (3) (4) (5) (6)  
 5276 \* 6.1.1. Signature Counter Considerations  
 5277 \* 6.2.1. The authenticatorMakeCredential operation (2) (3)  
 5278 \* 6.2.2. The authenticatorGetAssertion operation (2) (3) (4)  
 5279 \* 7.1. Registering a new credential (2)  
 5280 \* 7.2. Verifying an authentication assertion (2)  
 5281 \* 8.4. Android Key Attestation Statement Format  
 5282 \* 8.6. FIDO U2F Attestation Statement Format (2) (3)  
 5283  
 5284 #public-key-credentialReferenced in:  
 5285 \* 1. Introduction (2) (3) (4) (5)  
 5286 \* 4. Terminology (2) (3) (4) (5) (6) (7) (8)  
 5287 \* 5. Web Authentication API (2) (3) (4)  
 5288 \* 5.1. PublicKeyCredential Interface  
 5289 \* 5.1.3. 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)  
5067 \* 6.1. Registering a new credential  
5068 \* 8. WebAuthn Extensions (2)  
5069 \* 11. Sample scenarios

#test-of-user-presenceReferenced in:  
5070 \* 3. Terminology (2) (3) (4) (5) (6)  
5071 \* 9.2. Simple Transaction Authorization Extension (txAuthSimple)  
5072 \* 9.3. Generic Transaction Authorization Extension (txAuthGeneric)

#user-consentReferenced in:  
5073 \* 1. Introduction  
5074 \* 3. Terminology (2)  
5075 \* 4.1.3. [Create a new credential - PublicKeyCredential's](#)  
5076 [\[\[Create\]\]\(options\) method](#)  
5077 \* 4.2.2. [Web Authentication Assertion \(interface](#)  
5078 [AuthenticatorAssertionResponse\)](#)  
5079 \* 5. WebAuthn Authenticator model (2) (3)  
5080 \* 5.2.2. The authenticatorGetAssertion operation (2)

#user-verificationReferenced in:  
5081 \* 1. Introduction  
5082 \* 3. Terminology (2) (3) (4) (5) (6) (7) (8)  
5083 \* 4.1.3. [Create a new credential - PublicKeyCredential's](#)  
5084 [\[\[Create\]\]\(options\) method](#)  
5085 \* 9.2. Simple Transaction Authorization Extension (txAuthSimple)  
5086 \* 9.3. Generic Transaction Authorization Extension (txAuthGeneric)

#concept-user-presentReferenced in:  
5087 \* 3. Terminology  
5088 \* 5.1. Authenticator data (2) (3)

#upReferenced in:  
5089 \* 5.1. Authenticator data

#concept-user-verifiedReferenced in:  
5090 \* 3. Terminology  
5091 \* 5.1. Authenticator data (2) (3)

5290 [[Create]](options) method  
5291 \* 5.1.4. Use an existing credential to make an assertion  
5292 \* 5.1.4.1. PublicKeyCredential's  
5293 [\[\[DiscoverFromExternalSource\]\]\(options\) method](#)  
5294 \* 5.2.1. Information about Public Key Credential (interface  
5295 AuthenticatorAttestationResponse)  
5296 \* 5.4.1. Public Key Entity Description (dictionary  
5297 PublicKeyCredentialEntity)  
5298 \* 5.4.4. Authenticator Selection Criteria (dictionary  
5299 AuthenticatorSelectionCriteria)  
5300 \* 5.5. Options for Assertion Generation (dictionary  
5301 PublicKeyCredentialRequestOptions)  
5302 \* 5.7. Supporting Data Structures  
5303 \* 6. WebAuthn Authenticator model (2) (3) (4) (5)  
5304 \* 6.2.2. The authenticatorGetAssertion operation (2) (3)  
5305 \* 6.3. Attestation (2)  
5306 \* 6.3.2. Attestation Statement Formats  
5307 \* 6.3.3. Attestation Types  
5308 \* 6.3.5.2. [Attestation Certificate and Attestation Certificate CA](#)  
5309 Compromise (2)  
5310 \* 7.1. Registering a new credential  
5311 \* 9. WebAuthn Extensions (2)  
5312 \* 12. Sample scenarios

#test-of-user-presenceReferenced in:  
5313 \* 4. Terminology (2) (3) (4) (5) (6)  
5314 \* 10.2. Simple Transaction Authorization Extension (txAuthSimple)  
5315 \* 10.3. Generic Transaction Authorization Extension (txAuthGeneric)

#user-consentReferenced in:  
5316 \* 1. Introduction  
5317 \* 4. Terminology (2)  
5318 \* 5.2.2. [Web Authentication Assertion \(interface](#)  
5319 [AuthenticatorAssertionResponse\)](#)  
5320 \* 6. WebAuthn Authenticator model (2) (3)  
5321 \* 6.2.2. The authenticatorGetAssertion operation (2)

#user-handleReferenced in:  
5322 \* 5.1.4.1. PublicKeyCredential's  
5323 [\[\[DiscoverFromExternalSource\]\]\(options\) method](#)  
5324 \* 5.2.2. [Web Authentication Assertion \(interface](#)  
5325 [AuthenticatorAssertionResponse\)](#)  
5326 \* 5.4. Options for Credential Creation (dictionary  
5327 [MakePublicKeyCredentialOptions\)](#)  
5328 \* 5.4.3. [User Account Parameters for Credential Generation](#)  
5329 (dictionary PublicKeyCredentialUserEntity)  
5330 \* 6.2.1. The authenticatorMakeCredential operation  
5331 \* 6.2.2. The authenticatorGetAssertion operation

#user-verificationReferenced in:  
5332 \* 1. Introduction  
5333 \* 4. Terminology (2) (3) (4) (5) (6) (7) (8) (9)  
5334 \* 5.1.3. [Create a new credential - PublicKeyCredential's](#)  
5335 [\[\[Create\]\]\(options\) method](#)  
5336 \* 10.2. Simple Transaction Authorization Extension (txAuthSimple)  
5337 \* 10.3. Generic Transaction Authorization Extension (txAuthGeneric)

#concept-user-presentReferenced in:  
5338 \* 4. Terminology  
5339 \* 6.1. Authenticator data (2) (3)

#upReferenced in:  
5340 \* 6.1. Authenticator data

#concept-user-verifiedReferenced in:  
5341 \* 4. Terminology  
5342 \* 6.1. Authenticator data (2) (3)

```

5104
5105 #uvReferenced in:
5106 * 5.1. Authenticator data
5107
5108 #webauthn-clientReferenced in:
5109 * 3. Terminology (2)
5110
5111 #web-authentication-apiReferenced in:
5112 * 1. Introduction (2) (3)
5113 * 3. Terminology (2)
5114
5115 #publickeycredentialReferenced in:
5116 * 1. Introduction
5117 * 4.1. PublicKeyCredential Interface (2) (3) (4) (5) (6) (7) (8)
5118 * 4.1.3. Create a new credential - PublicKeyCredential's
5119 [[Create]](options) method (2) (3) (4) (5) (6)
5120 * 4.1.4. Use an existing credential to make an assertion -
5121 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5122 method (2) (3)
5123 * 4.1.5. Platform Authenticator Availability - PublicKeyCredential's
5124 isPlatformAuthenticatorAvailable() method
5125 * 4.7.3. Credential Descriptor (dictionary
5126 PublicKeyCredentialDescriptor)
5127 * 5.2.1. The authenticatorMakeCredential operation
5128 * 6. Relying Party Operations
5129 * 6.2. Verifying an authentication assertion
5130
5131 #dom-publickeycredential-rawidReferenced in:
5132 * 4.1. PublicKeyCredential Interface
5133 * 6.2. Verifying an authentication assertion
5134
5135 #dom-publickeycredential-responseReferenced in:
5136 * 4.1. PublicKeyCredential Interface
5137 * 4.1.3. Create a new credential - PublicKeyCredential's
5138 [[Create]](options) method
5139 * 4.1.4. Use an existing credential to make an assertion -
5140 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5141 method
5142 * 6.2. Verifying an authentication assertion
5143
5144 #dom-publickeycredential-clientextensionresultsReferenced in:
5145 * 4.1. PublicKeyCredential Interface
5146 * 4.1.3. Create a new credential - PublicKeyCredential's
5147 [[Create]](options) method
5148 * 4.1.4. Use an existing credential to make an assertion -
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

```

```

5357
5358 #uvReferenced in:
5359 * 6.1. Authenticator data
5360
5361 #webauthn-clientReferenced in:
5362 * 4. Terminology (2)
5363
5364 #web-authentication-apiReferenced in:
5365 * 1. Introduction (2) (3)
5366 * 4. Terminology (2)
5367
5368 #publickeycredentialReferenced in:
5369 * 1. Introduction
5370 * 5.1. PublicKeyCredential Interface (2) (3) (4) (5) (6) (7) (8)
5371 * 5.1.3. Create a new credential - PublicKeyCredential's
5372 [[Create]](options) method (2) (3) (4) (5) (6)
5373 * 5.1.4.1. PublicKeyCredential's
5374 [[DiscoverFromExternalSource]](options) method (2) (3)
5375 * 5.1.5. Store an existing credential - PublicKeyCredential's
5376 [[Store]](credential) method (2)
5377 * 5.1.6. Platform Authenticator Availability - PublicKeyCredential's
5378 isPlatformAuthenticatorAvailable() method
5379 * 5.7.3. Credential Descriptor (dictionary
5380 PublicKeyCredentialDescriptor)
5381 * 7. Relying Party Operations
5382 * 7.2. Verifying an authentication assertion
5383
5384 #dom-publickeycredential-rawidReferenced in:
5385 * 5.1. PublicKeyCredential Interface
5386 * 7.2. Verifying an authentication assertion
5387
5388 #dom-publickeycredential-responseReferenced in:
5389 * 5.1. PublicKeyCredential Interface
5390 * 5.1.3. Create a new credential - PublicKeyCredential's
5391 [[Create]](options) method
5392 * 5.1.4.1. PublicKeyCredential's
5393 [[DiscoverFromExternalSource]](options) method
5394 * 7.2. Verifying an authentication assertion
5395
5396 #dom-publickeycredential-clientextensionresultsReferenced in:
5397 * 5.1. PublicKeyCredential Interface
5398 * 5.1.3. Create a new credential - PublicKeyCredential's
5399 [[Create]](options) method
5400 * 5.1.4.1. PublicKeyCredential's
5401 [[DiscoverFromExternalSource]](options) method
5402 * 9.4. Client extension processing
5403
5404 #dom-publickeycredential-identifier-slotReferenced in:
5405 * 5.1. PublicKeyCredential Interface (2)
5406 * 5.1.3. Create a new credential - PublicKeyCredential's
5407 [[Create]](options) method
5408 * 5.1.4.1. PublicKeyCredential's
5409 [[DiscoverFromExternalSource]](options) method
5410
5411 #dom-credentialcreationoptions-publickeyReferenced in:
5412 * 5.1.3. Create a new credential - PublicKeyCredential's
5413 [[Create]](options) method (2) (3)
5414
5415 #dom-credentialrequestoptions-publickeyReferenced in:
5416 * 5.1.4.1. PublicKeyCredential's
5417 [[DiscoverFromExternalSource]](options) method (2)
5418
5419 #dom-publickeycredential-create-slotReferenced in:
5420 * 5.1. PublicKeyCredential Interface
5421

```

5175 #dom-publickeycredential-create-options-optionsReferenced in:  
5176 \* 6.1. Registering a new credential

5175 #dom-publickeycredential-discoverfromexternalsource-slotReferenced in:  
5176 \* 4.1. PublicKeyCredential Interface

5175 #authenticatorresponseReferenced in:  
5176 \* 4.1. PublicKeyCredential Interface (2)  
5180 \* 4.2. Authenticator Responses (interface AuthenticatorResponse) (2)  
5181 \* 4.2.1. Information about Public Key Credential (interface  
5182 AuthenticatorAttestationResponse) (2)  
5183 \* 4.2.2. Web Authentication Assertion (interface  
5184 AuthenticatorAssertionResponse) (2)

5175 #dom-authenticatorresponse-clientdatajsonReferenced in:  
5176 \* 4.1.3. Create a new credential - PublicKeyCredential's  
5177 [[Create]](options) method (2)  
5178 \* 4.1.4. Use an existing credential to make an assertion -  
5179 PublicKeyCredential's [[DiscoverFromExternalSource]](options)  
5180 method (2)  
5181 \* 4.2. Authenticator Responses (interface AuthenticatorResponse)  
5182 \* 4.2.1. Information about Public Key Credential (interface  
5183 AuthenticatorAttestationResponse)  
5184 \* 4.2.2. Web Authentication Assertion (interface  
5185 AuthenticatorAssertionResponse)  
5186 \* 6.1. Registering a new credential (2)  
5187 \* 6.2. Verifying an authentication assertion

5175 #authenticatorattestationresponseReferenced in:  
5176 \* 4.1. PublicKeyCredential Interface  
5177 \* 4.1.3. Create a new credential - PublicKeyCredential's  
5178 [[Create]](options) method (2)  
5179 \* 4.2.1. Information about Public Key Credential (interface  
5180 AuthenticatorAttestationResponse) (2)  
5181 \* 6. Relying Party Operations  
5182 \* 6.1. Registering a new credential (2) (3)

5175 #dom-authenticatorattestationresponse-attestationobjectReferenced in:  
5176 \* 4.1.3. Create a new credential - PublicKeyCredential's  
5177 [[Create]](options) method  
5178 \* 4.2.1. Information about Public Key Credential (interface  
5179 AuthenticatorAttestationResponse)  
5180 \* 6.1. Registering a new credential

5175 #authenticatorassertionresponseReferenced in:  
5176 \* 3. Terminology  
5177 \* 4.1. PublicKeyCredential Interface  
5178 \* 4.1.4. Use an existing credential to make an assertion -  
5179 PublicKeyCredential's [[DiscoverFromExternalSource]](options)  
5180 method  
5181 \* 4.2.2. Web Authentication Assertion (interface  
5182 AuthenticatorAssertionResponse) (2)  
5183 \* 6. Relying Party Operations

5175 #dom-authenticatorassertionresponse-authenticatordataReferenced in:  
5176 \* 4.1.4. Use an existing credential to make an assertion -  
5177 PublicKeyCredential's [[DiscoverFromExternalSource]](options)  
5178 method (2)  
5179 \* 4.2.2. Web Authentication Assertion (interface  
5180 AuthenticatorAssertionResponse)  
5181 \* 6.2. Verifying an authentication assertion

5175 #dom-authenticatorassertionresponse-signatureReferenced in:  
5176 \* 4.1.4. Use an existing credential to make an assertion -  
5177 PublicKeyCredential's [[DiscoverFromExternalSource]](options)  
5178 method (2)

5422 #dom-publickeycredential-create-options-optionsReferenced in:  
5423 \* 7.1. Registering a new credential

5422 #dom-publickeycredential-collectfromcredentialstore-slotReferenced in:  
5423 \* 5.1.4. Use an existing credential to make an assertion

5422 #dom-publickeycredential-discoverfromexternalsource-slotReferenced in:  
5423 \* 5.1. PublicKeyCredential Interface  
5424 \* 5.1.4. Use an existing credential to make an assertion

5422 #authenticatorresponseReferenced in:  
5423 \* 5.1. PublicKeyCredential Interface (2)  
5424 \* 5.2. Authenticator Responses (interface AuthenticatorResponse) (2)  
5425 \* 5.2.1. Information about Public Key Credential (interface  
5426 AuthenticatorAttestationResponse) (2)  
5427 \* 5.2.2. Web Authentication Assertion (interface  
5428 AuthenticatorAssertionResponse) (2)

5422 #dom-authenticatorresponse-clientdatajsonReferenced in:  
5423 \* 5.1.3. Create a new credential - PublicKeyCredential's  
5424 [[Create]](options) method (2)  
5425 \* 5.1.4.1. PublicKeyCredential's  
5426 [[DiscoverFromExternalSource]](options) method (2)  
5427 \* 5.2. Authenticator Responses (interface AuthenticatorResponse)  
5428 \* 5.2.1. Information about Public Key Credential (interface  
5429 AuthenticatorAttestationResponse)  
5430 \* 5.2.2. Web Authentication Assertion (interface  
5431 AuthenticatorAssertionResponse)  
5432 \* 5.2.2. Web Authentication Assertion (interface  
5433 AuthenticatorAssertionResponse)  
5434 \* 7.1. Registering a new credential (2)  
5435 \* 7.2. Verifying an authentication assertion

5422 #authenticatorattestationresponseReferenced in:  
5423 \* 5.1. PublicKeyCredential Interface  
5424 \* 5.1.3. Create a new credential - PublicKeyCredential's  
5425 [[Create]](options) method (2)  
5426 \* 5.2.1. Information about Public Key Credential (interface  
5427 AuthenticatorAttestationResponse) (2)  
5428 \* 7. Relying Party Operations  
5429 \* 7.1. Registering a new credential (2) (3)

5422 #dom-authenticatorattestationresponse-attestationobjectReferenced in:  
5423 \* 5.1.3. Create a new credential - PublicKeyCredential's  
5424 [[Create]](options) method  
5425 \* 5.2.1. Information about Public Key Credential (interface  
5426 AuthenticatorAttestationResponse)  
5427 \* 7.1. Registering a new credential

5422 #authenticatorassertionresponseReferenced in:  
5423 \* 4. Terminology  
5424 \* 5.1. PublicKeyCredential Interface  
5425 \* 5.1.4.1. PublicKeyCredential's  
5426 [[DiscoverFromExternalSource]](options) method  
5427 \* 5.2.2. Web Authentication Assertion (interface  
5428 AuthenticatorAssertionResponse) (2)  
5429 \* 7. Relying Party Operations

5422 #dom-authenticatorassertionresponse-authenticatordataReferenced in:  
5423 \* 5.1.4.1. PublicKeyCredential's  
5424 [[DiscoverFromExternalSource]](options) method (2)  
5425 \* 5.2.2. Web Authentication Assertion (interface  
5426 AuthenticatorAssertionResponse)  
5427 \* 7.2. Verifying an authentication assertion

5422 #dom-authenticatorassertionresponse-signatureReferenced in:  
5423 \* 5.1.4.1. PublicKeyCredential's  
5424 [[DiscoverFromExternalSource]](options) method (2)  
5425 \* 5.2.2. Web Authentication Assertion (interface



5235 \* 4.2.2. Web Authentication Assertion (interface)

5240 AuthenticatorAssertionResponse)

5241 \* 6.2. Verifying an authentication assertion

5242

5243 #dictdef-publickeycredentialparametersReferenced in:

5244 \* 4.3. Parameters for Credential Generation (dictionary

5245 PublicKeyCredentialParameters)

5246 \* 4.4. Options for Credential Creation (dictionary

5247 MakePublicKeyCredentialOptions) (2)

5248

5249 #dom-publickeycredentialparameters-typeReferenced in:

5250 \* 4.1.3. Create a new credential - PublicKeyCredential's

5251 [[Create]](options) method (2)

5252 \* 4.3. Parameters for Credential Generation (dictionary

5253 PublicKeyCredentialParameters)

5254

5255 #dom-publickeycredentialparameters-algReferenced in:

5256 \* 4.1.3. Create a new credential - PublicKeyCredential's

5257 [[Create]](options) method

5258 \* 4.3. Parameters for Credential Generation (dictionary

5259 PublicKeyCredentialParameters)

5260

5261 #dictdef-makepublickeycredentialoptionsReferenced in:

5262 \* 4.1.1. CredentialCreationOptions Extension

5263 \* 4.1.3. Create a new credential - PublicKeyCredential's

5264 [[Create]](options) method

5265 \* 4.4. Options for Credential Creation (dictionary

5266 MakePublicKeyCredentialOptions)

5267

5268 #dom-makepublickeycredentialoptions-rpReferenced in:

5269 \* 4.1.3. Create a new credential - PublicKeyCredential's

5270 [[Create]](options) method (2) (3) (4) (5) (6)

5271 \* 4.4. Options for Credential Creation (dictionary

5272 MakePublicKeyCredentialOptions)

5273

5274 #dom-makepublickeycredentialoptions-userReferenced in:

5275 \* 4.1.3. Create a new credential - PublicKeyCredential's

5276 [[Create]](options) method (2) (3) (4)

5277 \* 4.4. Options for Credential Creation (dictionary

5278 MakePublicKeyCredentialOptions)

5279 \* 5.2.1. The authenticatorMakeCredential operation (2)

5280 \* 6.1. Registering a new credential

5281

5282 #dom-makepublickeycredentialoptions-challengeReferenced in:

5283 \* 4.1.3. Create a new credential - PublicKeyCredential's

5284 [[Create]](options) method

5285 \* 4.4. Options for Credential Creation (dictionary

5286 MakePublicKeyCredentialOptions)

5287

5288 #dom-makepublickeycredentialoptions-pubkeycredparamsReferenced in:

5289 \* 4.1.3. Create a new credential - PublicKeyCredential's

5290 [[Create]](options) method (2)

5291 \* 4.4. Options for Credential Creation (dictionary

5292 MakePublicKeyCredentialOptions)

5293

5294 #dom-makepublickeycredentialoptions-timeoutReferenced in:

5295 \* 4.1.3. Create a new credential - PublicKeyCredential's

5296 [[Create]](options) method (2)

5297 \* 4.4. Options for Credential Creation (dictionary

5298 MakePublicKeyCredentialOptions)

5299

5300 #dom-makepublickeycredentialoptions-excludecredentialsReferenced in:

5301 \* 4.1.3. Create a new credential - PublicKeyCredential's

5302 [[Create]](options) method

5489 AuthenticatorAssertionResponse)

5490 \* 7.2. Verifying an authentication assertion

5491

5492 #dom-authenticatorassertionresponse-userhandleReferenced in:

5493 \* 5.1.4.1. PublicKeyCredential's

5494 [[DiscoverFromExternalSource]](options) method

5495 \* 5.2.2. Web Authentication Assertion (interface

5496 AuthenticatorAssertionResponse)

5497

5498 #dictdef-publickeycredentialparametersReferenced in:

5499 \* 5.3. Parameters for Credential Generation (dictionary

5500 PublicKeyCredentialParameters)

5501 \* 5.4. Options for Credential Creation (dictionary

5502 MakePublicKeyCredentialOptions) (2)

5503

5504 #dom-publickeycredentialparameters-typeReferenced in:

5505 \* 5.1.3. Create a new credential - PublicKeyCredential's

5506 [[Create]](options) method (2)

5507 \* 5.3. Parameters for Credential Generation (dictionary

5508 PublicKeyCredentialParameters)

5509

5510 #dom-publickeycredentialparameters-algReferenced in:

5511 \* 5.1.3. Create a new credential - PublicKeyCredential's

5512 [[Create]](options) method

5513 \* 5.3. Parameters for Credential Generation (dictionary

5514 PublicKeyCredentialParameters)

5515

5516 #dictdef-makepublickeycredentialoptionsReferenced in:

5517 \* 5.1.1. CredentialCreationOptions Extension

5518 \* 5.1.3. Create a new credential - PublicKeyCredential's

5519 [[Create]](options) method

5520 \* 5.4. Options for Credential Creation (dictionary

5521 MakePublicKeyCredentialOptions)

5522

5523 #dom-makepublickeycredentialoptions-rpReferenced in:

5524 \* 5.1.3. Create a new credential - PublicKeyCredential's

5525 [[Create]](options) method (2) (3) (4) (5) (6)

5526 \* 5.4. Options for Credential Creation (dictionary

5527 MakePublicKeyCredentialOptions)

5528

5529 #dom-makepublickeycredentialoptions-userReferenced in:

5530 \* 5.1.3. Create a new credential - PublicKeyCredential's

5531 [[Create]](options) method (2) (3) (4)

5532 \* 5.4. Options for Credential Creation (dictionary

5533 MakePublicKeyCredentialOptions)

5534 \* 7.1. Registering a new credential

5535

5536 #dom-makepublickeycredentialoptions-challengeReferenced in:

5537 \* 5.1.3. Create a new credential - PublicKeyCredential's

5538 [[Create]](options) method

5539 \* 5.4. Options for Credential Creation (dictionary

5540 MakePublicKeyCredentialOptions)

5541

5542 #dom-makepublickeycredentialoptions-pubkeycredparamsReferenced in:

5543 \* 5.1.3. Create a new credential - PublicKeyCredential's

5544 [[Create]](options) method (2)

5545 \* 5.4. Options for Credential Creation (dictionary

5546 MakePublicKeyCredentialOptions)

5547

5548 #dom-makepublickeycredentialoptions-timeoutReferenced in:

5549 \* 5.1.3. Create a new credential - PublicKeyCredential's

5550 [[Create]](options) method (2)

5551 \* 5.4. Options for Credential Creation (dictionary

5552 MakePublicKeyCredentialOptions)

5553

5554 #dom-makepublickeycredentialoptions-excludecredentialsReferenced in:

5555 \* 5.1.3. Create a new credential - PublicKeyCredential's

5556 [[Create]](options) method

5303 \* 4.4. Options for Credential Creation (dictionary  
5304 MakePublicKeyCredentialOptions)  
5305  
5306 #dom-makepublickeycredentialoptions-authenticatorselectionReferenced  
5307 in:  
5308 \* 4.1.3. Create a new credential - PublicKeyCredential's  
5309 [[Create]](options) method (2)  
5310 \* 4.4. Options for Credential Creation (dictionary  
5311 MakePublicKeyCredentialOptions)  
5312 \* 5.2.1. The authenticatorMakeCredential operation (2)  
5313  
5314 #dom-makepublickeycredentialoptions-extensionsReferenced in:  
5315 \* 4.1.3. Create a new credential - PublicKeyCredential's  
5316 [[Create]](options) method (2)  
5317 \* 4.4. Options for Credential Creation (dictionary  
5318 MakePublicKeyCredentialOptions)  
5319 \* 8.3. Extending request parameters  
5320  
5321 #dictdef-publickeycredentialentityReferenced in:  
5322 \* 4.4. Options for Credential Creation (dictionary  
5323 MakePublicKeyCredentialOptions) (2)  
5324 \* 4.4.1. Public Key Entity Description (dictionary  
5325 PublicKeyCredentialEntity) (2)  
5326 \* 4.4.2. User Account Parameters for Credential Generation  
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)

5557 \* 5.4. Options for Credential Creation (dictionary  
5558 MakePublicKeyCredentialOptions)  
5559  
5560 #dom-makepublickeycredentialoptions-authenticatorselectionReferenced  
5561 in:  
5562 \* 5.1.3. Create a new credential - PublicKeyCredential's  
5563 [[Create]](options) method (2)  
5564 \* 5.4. Options for Credential Creation (dictionary  
5565 MakePublicKeyCredentialOptions)  
5566 \* 6.2.1. The authenticatorMakeCredential operation (2)  
5567  
5568 #dom-makepublickeycredentialoptions-extensionsReferenced in:  
5569 \* 5.1.3. Create a new credential - PublicKeyCredential's  
5570 [[Create]](options) method (2)  
5571 \* 5.4. Options for Credential Creation (dictionary  
5572 MakePublicKeyCredentialOptions)  
5573 \* 9.3. Extending request parameters  
5574  
5575 #dictdef-publickeycredentialentityReferenced in:  
5576 \* 5.4.1. Public Key Entity Description (dictionary  
5577  
5578  
5579  
5580  
5581  
5582  
5583  
5584  
5585  
5586  
5587  
5588  
5589  
5590  
5591  
5592  
5593  
5594  
5595  
5596  
5597  
5598  
5599  
5600  
5601  
5602  
5603  
5604  
5605  
5606  
5607  
5608  
5609  
5610  
5611  
5612  
5613  
5614

PublicKeyCredentialEntity) (2)  
\* 5.4.2. RP Parameters for Credential Generation (dictionary  
PublicKeyCredentialRpEntity)  
\* 5.4.3. User Account Parameters for Credential Generation  
(dictionary PublicKeyCredentialUserEntity)

#dom-publickeycredentialentity-nameReferenced in:  
\* 5.1.3. Create a new credential - PublicKeyCredential's  
[[Create]](options) method (2)  
\* 5.4. Options for Credential Creation (dictionary  
MakePublicKeyCredentialOptions) (2)  
\* 5.4.1. Public Key Entity Description (dictionary  
PublicKeyCredentialEntity)

#dom-publickeycredentialentity-iconReferenced in:  
\* 5.4.1. Public Key Entity Description (dictionary  
PublicKeyCredentialEntity)

#dictdef-publickeycredentialrpentityReferenced in:  
\* 5.4. Options for Credential Creation (dictionary  
MakePublicKeyCredentialOptions) (2)  
\* 5.4.2. RP Parameters for Credential Generation (dictionary  
PublicKeyCredentialRpEntity) (2)  
\* 6.2.1. The authenticatorMakeCredential operation

#dom-publickeycredentialrpentity-idReferenced in:  
\* 5.1.3. Create a new credential - PublicKeyCredential's  
[[Create]](options) method (2) (3) (4)  
\* 5.4. Options for Credential Creation (dictionary  
MakePublicKeyCredentialOptions)  
\* 5.4.2. RP Parameters for Credential Generation (dictionary  
PublicKeyCredentialRpEntity)

#dictdef-publickeycredentialuserentityReferenced in:  
\* 5.4. Options for Credential Creation (dictionary  
MakePublicKeyCredentialOptions) (2)  
\* 5.4.3. User Account Parameters for Credential Generation  
(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
5382 AuthenticatorSelectionCriteria)
5383
5384 #dom-authenticatorselectioncriteria-uvReferenced in:
5385 * 4.1.3. Create a new credential - PublicKeyCredential's
5386 [[Create]](options) method
5387 * 4.4.3. Authenticator Selection Criteria (dictionary
5388 AuthenticatorSelectionCriteria)
5389
5390 #enumdef-authenticatorattachmentReferenced in:
5391 * 4.4.3. Authenticator Selection Criteria (dictionary
5392 AuthenticatorSelectionCriteria) (2)
5393 * 4.4.4. Authenticator Attachment enumeration (enum
5394 AuthenticatorAttachment) (2)
5395
5396 #platform-authenticatorsReferenced in:
5397 * 4.1.5. Platform Authenticator Availability - PublicKeyCredential's
5398 isPlatformAuthenticatorAvailable() method (2) (3) (4) (5)
5399 * 4.4.4. Authenticator Attachment enumeration (enum
5400 AuthenticatorAttachment) (2)
5401 * 11.1. Registration
5402 * 11.2. Registration Specifically with Platform Authenticator (2)
5403
5404 #roaming-authenticatorsReferenced in:
5405 * 1.1.3. Other use cases and configurations
5406 * 4.4.4. Authenticator Attachment enumeration (enum
5407 AuthenticatorAttachment) (2)
5408 * 11.1. Registration
5409
5410 #platform-attachmentReferenced in:
5411 * 4.4.4. Authenticator Attachment enumeration (enum

```

```

5615 * 6.2.1. The authenticatorMakeCredential operation
5616
5617 #dom-publickeycredentialuserentity-idReferenced in:
5618 * 5.1.3. Create a new credential - PublicKeyCredential's
5619 [[Create]](options) method
5620 * 5.4. Options for Credential Creation (dictionary
5621 MakePublicKeyCredentialOptions)
5622 * 5.4.3. User Account Parameters for Credential Generation
5623 (dictionary PublicKeyCredentialUserEntity)
5624 * 6.2.1. The authenticatorMakeCredential operation
5625
5626 #dom-publickeycredentialuserentity-displaynameReferenced in:
5627 * 5.1.3. Create a new credential - PublicKeyCredential's
5628 [[Create]](options) method
5629 * 5.4. Options for Credential Creation (dictionary
5630 MakePublicKeyCredentialOptions)
5631 * 5.4.3. User Account Parameters for Credential Generation
5632 (dictionary PublicKeyCredentialUserEntity)
5633
5634 #dictdef-authenticatorselectioncriteriaReferenced in:
5635 * 5.4. Options for Credential Creation (dictionary
5636 MakePublicKeyCredentialOptions) (2)
5637 * 5.4.4. Authenticator Selection Criteria (dictionary
5638 AuthenticatorSelectionCriteria) (2)
5639
5640 #dom-authenticatorselectioncriteria-authenticatorattachmentReferenced
5641 in:
5642 * 5.1.3. Create a new credential - PublicKeyCredential's
5643 [[Create]](options) method
5644 * 5.4.4. Authenticator Selection Criteria (dictionary
5645 AuthenticatorSelectionCriteria)
5646
5647 #dom-authenticatorselectioncriteria-requireresidentkeyReferenced in:
5648 * 5.1.3. Create a new credential - PublicKeyCredential's
5649 [[Create]](options) method (2)
5650 * 5.4.4. Authenticator Selection Criteria (dictionary
5651 AuthenticatorSelectionCriteria)
5652 * 6.2.1. The authenticatorMakeCredential operation
5653
5654 #dom-authenticatorselectioncriteria-requireuserverificationReferenced
5655 in:
5656 * 5.1.3. Create a new credential - PublicKeyCredential's
5657 [[Create]](options) method
5658 * 5.4.4. Authenticator Selection Criteria (dictionary
5659 AuthenticatorSelectionCriteria)
5660 * 6.2.1. The authenticatorMakeCredential operation
5661
5662 #enumdef-authenticatorattachmentReferenced in:
5663 * 5.4.4. Authenticator Selection Criteria (dictionary
5664 AuthenticatorSelectionCriteria) (2)
5665 * 5.4.5. Authenticator Attachment enumeration (enum
5666 AuthenticatorAttachment) (2)
5667
5668 #platform-authenticatorsReferenced in:
5669 * 5.1.6. Platform Authenticator Availability - PublicKeyCredential's
5670 isPlatformAuthenticatorAvailable() method (2) (3) (4) (5)
5671 * 5.4.5. Authenticator Attachment enumeration (enum
5672 AuthenticatorAttachment) (2)
5673 * 6.2.1. The authenticatorMakeCredential operation
5674 * 12.1. Registration
5675 * 12.2. Registration Specifically with Platform Authenticator (2)
5676
5677 #roaming-authenticatorsReferenced in:
5678 * 1.1.3. Other use cases and configurations
5679 * 5.4.5. Authenticator Attachment enumeration (enum
5680 AuthenticatorAttachment) (2)
5681 * 12.1. Registration
5682
5683 #platform-attachmentReferenced in:
5684 * 5.4.5. Authenticator Attachment enumeration (enum

```



5412 AuthenticatorAttachment)  
5413  
5414 #cross-platform-attachedReferenced in:  
5415 \* 4.4.4. Authenticator Attachment enumeration (enum  
5416 AuthenticatorAttachment) (2)  
5417  
5418 #dictdef-publickeycredentialrequestoptionsReferenced in:  
5419 \* 4.1.2. CredentialRequestOptions Extension  
5420 \* 4.5. Options for Assertion Generation (dictionary  
5421 PublicKeyCredentialRequestOptions) (2)  
5422 \* 6.2. Verifying an authentication assertion  
5423  
5424 #dom-publickeycredentialrequestoptions-challengeReferenced in:  
5425 \* 4.1.4. Use an existing credential to make an assertion -  
5426 PublicKeyCredential's [\[\[DiscoverFromExternalSource\]\]\(options\)](#)  
5427 method  
5428 \* 4.5. Options for Assertion Generation (dictionary  
5429 PublicKeyCredentialRequestOptions) (2)  
5430  
5431 #dom-publickeycredentialrequestoptions-timeoutReferenced in:  
5432 \* 4.1.4. Use an existing credential to make an assertion -  
5433 PublicKeyCredential's [\[\[DiscoverFromExternalSource\]\]\(options\)](#)  
5434 method (2)  
5435 \* 4.5. Options for Assertion Generation (dictionary  
5436 PublicKeyCredentialRequestOptions)  
5437  
5438 #dom-publickeycredentialrequestoptions-rpidReferenced in:  
5439 \* 4.1.4. Use an existing credential to make an assertion -  
5440 PublicKeyCredential's [\[\[DiscoverFromExternalSource\]\]\(options\)](#)  
5441 method (2) (3) (4)  
5442 \* 4.5. Options for Assertion Generation (dictionary  
5443 PublicKeyCredentialRequestOptions)  
5444 \* 9.1. FIDO AppId Extension (appid)  
5445  
5446 #dom-publickeycredentialrequestoptions-allowcredentialsReferenced in:  
5447 \* 4.1.4. Use an existing credential to make an assertion -  
5448 PublicKeyCredential's [\[\[DiscoverFromExternalSource\]\]\(options\)](#)  
5449 method (2) (3) (4)  
5450 \* 4.5. Options for Assertion Generation (dictionary  
5451 PublicKeyCredentialRequestOptions)  
5452  
5453 #dom-publickeycredentialrequestoptions-extensionsReferenced in:  
5454 \* 4.1.4. Use an existing credential to make an assertion -  
5455 PublicKeyCredential's [\[\[DiscoverFromExternalSource\]\]\(options\)](#)  
5456 method (2)  
5457 \* 4.5. Options for Assertion Generation (dictionary  
5458 PublicKeyCredentialRequestOptions)  
5459  
5460 #typedefdef-authenticationextensionsReferenced in:  
5461 \* 4.1. PublicKeyCredential Interface (2)  
5462 \* 4.1.3. Create a new credential - PublicKeyCredential's  
5463 [\[\[Create\]\]\(options\)](#) method  
5464 \* 4.1.4. Use an existing credential to make an assertion -  
5465 PublicKeyCredential's [\[\[DiscoverFromExternalSource\]\]\(options\)](#)  
5466 method  
5467 \* 4.4. Options for Credential Creation (dictionary  
5468 MakePublicKeyCredentialOptions) (2)  
5469 \* 4.5. Options for Assertion Generation (dictionary  
5470 PublicKeyCredentialRequestOptions) (2)  
5471 \* 4.7.1. Client data used in WebAuthn signatures (dictionary  
5472 CollectedClientData) (2)  
5473  
5474 #dictdef-collectedclientdataReferenced in:  
5475 \* 4.1.3. Create a new credential - PublicKeyCredential's  
5476 [\[\[Create\]\]\(options\)](#) method  
5477 \* 4.1.4. Use an existing credential to make an assertion -  
5478 PublicKeyCredential's [\[\[DiscoverFromExternalSource\]\]\(options\)](#)  
5479 method  
5480 \* 4.7.1. Client data used in WebAuthn signatures (dictionary  
5481 CollectedClientData) (2)

5685 AuthenticatorAttachment)  
5686  
5687 #cross-platform-attachedReferenced in:  
5688 \* 5.4.5. Authenticator Attachment enumeration (enum  
5689 AuthenticatorAttachment) (2)  
5690  
5691 #dictdef-publickeycredentialrequestoptionsReferenced in:  
5692 \* 5.1.2. CredentialRequestOptions Extension  
5693 \* 5.5. Options for Assertion Generation (dictionary  
5694 PublicKeyCredentialRequestOptions) (2)  
5695 \* 7.2. Verifying an authentication assertion  
5696  
5697 #dom-publickeycredentialrequestoptions-challengeReferenced in:  
5698 \* 5.1.4.1. PublicKeyCredential's  
5699 [\[\[DiscoverFromExternalSource\]\]\(options\)](#) method  
5700 \* 5.5. Options for Assertion Generation (dictionary  
5701 PublicKeyCredentialRequestOptions) (2)  
5702  
5703 #dom-publickeycredentialrequestoptions-timeoutReferenced in:  
5704 \* 5.1.4.1. PublicKeyCredential's  
5705 [\[\[DiscoverFromExternalSource\]\]\(options\)](#) method (2)  
5706 \* 5.5. Options for Assertion Generation (dictionary  
5707 PublicKeyCredentialRequestOptions)  
5708  
5709 #dom-publickeycredentialrequestoptions-rpidReferenced in:  
5710 \* 5.1.4.1. PublicKeyCredential's  
5711 [\[\[DiscoverFromExternalSource\]\]\(options\)](#) method (2) (3) (4)  
5712 \* 5.5. Options for Assertion Generation (dictionary  
5713 PublicKeyCredentialRequestOptions)  
5714 \* 10.1. FIDO AppId Extension (appid)  
5715  
5716 #dom-publickeycredentialrequestoptions-allowcredentialsReferenced in:  
5717 \* 5.1.4.1. PublicKeyCredential's  
5718 [\[\[DiscoverFromExternalSource\]\]\(options\)](#) method (2) (3) (4)  
5719 \* 5.5. Options for Assertion Generation (dictionary  
5720 PublicKeyCredentialRequestOptions)  
5721  
5722 #dom-publickeycredentialrequestoptions-extensionsReferenced in:  
5723 \* 5.1.4.1. PublicKeyCredential's  
5724 [\[\[DiscoverFromExternalSource\]\]\(options\)](#) method (2)  
5725 \* 5.5. Options for Assertion Generation (dictionary  
5726 PublicKeyCredentialRequestOptions)  
5727  
5728 #typedefdef-authenticationextensionsReferenced in:  
5729 \* 5.1. PublicKeyCredential Interface (2)  
5730 \* 5.1.3. Create a new credential - PublicKeyCredential's  
5731 [\[\[Create\]\]\(options\)](#) method  
5732 \* 5.1.4.1. PublicKeyCredential's  
5733 [\[\[DiscoverFromExternalSource\]\]\(options\)](#) method  
5734 \* 5.4. Options for Credential Creation (dictionary  
5735 MakePublicKeyCredentialOptions) (2)  
5736 \* 5.5. Options for Assertion Generation (dictionary  
5737 PublicKeyCredentialRequestOptions) (2)  
5738 \* 5.7.1. Client data used in WebAuthn signatures (dictionary  
5739 CollectedClientData) (2)  
5740  
5741 #dictdef-collectedclientdataReferenced in:  
5742 \* 5.1.3. Create a new credential - PublicKeyCredential's  
5743 [\[\[Create\]\]\(options\)](#) method  
5744 \* 5.1.4.1. PublicKeyCredential's  
5745 [\[\[DiscoverFromExternalSource\]\]\(options\)](#) method  
5746 \* 5.7.1. Client data used in WebAuthn signatures (dictionary  
5747 CollectedClientData) (2)

```

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
5490 * 8.4. Client extension processing
5491 * 8.6. Example Extension
5492
5493 #dom-collectedclientdata-challengeReferenced in:
5494 * 4.1.3. Create a new credential - PublicKeyCredential's
5495 [[Create]](options) method
5496 * 4.1.4. Use an existing credential to make an assertion -
5497 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5498 method
5499 * 4.7.1. Client data used in WebAuthn signatures (dictionary
5500 CollectedClientData)
5501 * 6.1. Registering a new credential
5502 * 6.2. Verifying an authentication assertion
5503
5504 #dom-collectedclientdata-originReferenced in:
5505 * 4.1.3. Create a new credential - PublicKeyCredential's
5506 [[Create]](options) method
5507 * 4.1.4. Use an existing credential to make an assertion -
5508 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5509 method
5510 * 4.7.1. Client data used in WebAuthn signatures (dictionary
5511 CollectedClientData)
5512 * 6.1. Registering a new credential
5513 * 6.2. Verifying an authentication assertion
5514
5515 #dom-collectedclientdata-hashalgorithmReferenced in:
5516 * 4.1.3. Create a new credential - PublicKeyCredential's
5517 [[Create]](options) method
5518 * 4.1.4. Use an existing credential to make an assertion -
5519 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5520 method
5521 * 4.7.1. Client data used in WebAuthn signatures (dictionary
5522 CollectedClientData) (2)
5523 * 6.1. Registering a new credential
5524 * 6.2. Verifying an authentication assertion
5525
5526 #dom-collectedclientdata-tokenbindingidReferenced in:
5527 * 4.1.3. Create a new credential - PublicKeyCredential's
5528 [[Create]](options) method
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

```

```

5748 #client-dataReferenced in:
5749 * 5.2. Authenticator Responses (interface AuthenticatorResponse)
5750 * 6. WebAuthn Authenticator model (2) (3) (4)
5751 * 6.1. Authenticator data (2)
5752 * 7.1. Registering a new credential
5753 * 7.2. Verifying an authentication assertion
5754 * 9. WebAuthn Extensions
5755 * 9.4. Client extension processing
5756 * 9.6. Example Extension
5757
5758 #dom-collectedclientdata-challengeReferenced in:
5759 * 5.1.3. Create a new credential - PublicKeyCredential's
5760 [[Create]](options) method
5761 * 5.1.4.1. PublicKeyCredential's
5762 [[DiscoverFromExternalSource]](options) method
5763 * 5.7.1. Client data used in WebAuthn signatures (dictionary
5764 CollectedClientData)
5765 * 7.1. Registering a new credential
5766 * 7.2. Verifying an authentication assertion
5767
5768 #dom-collectedclientdata-originReferenced in:
5769 * 5.1.3. Create a new credential - PublicKeyCredential's
5770 [[Create]](options) method
5771 * 5.1.4.1. PublicKeyCredential's
5772 [[DiscoverFromExternalSource]](options) method
5773 * 5.7.1. Client data used in WebAuthn signatures (dictionary
5774 CollectedClientData)
5775 * 7.1. Registering a new credential
5776 * 7.2. Verifying an authentication assertion
5777
5778 #dom-collectedclientdata-hashalgorithmReferenced in:
5779 * 5.1.3. Create a new credential - PublicKeyCredential's
5780 [[Create]](options) method
5781 * 5.1.4.1. PublicKeyCredential's
5782 [[DiscoverFromExternalSource]](options) method
5783 * 5.7.1. Client data used in WebAuthn signatures (dictionary
5784 CollectedClientData) (2)
5785 * 7.1. Registering a new credential
5786 * 7.2. Verifying an authentication assertion
5787
5788 #dom-collectedclientdata-tokenbindingidReferenced in:
5789 * 5.1.3. Create a new credential - PublicKeyCredential's
5790 [[Create]](options) method
5791 * 5.1.4.1. PublicKeyCredential's
5792 [[DiscoverFromExternalSource]](options) method
5793 * 5.7.1. Client data used in WebAuthn signatures (dictionary
5794 CollectedClientData)
5795 * 7.1. Registering a new credential
5796 * 7.2. Verifying an authentication assertion
5797
5798 #dom-collectedclientdata-clientextensionsReferenced in:
5799 * 5.1.3. Create a new credential - PublicKeyCredential's
5800 [[Create]](options) method
5801 * 5.1.4.1. PublicKeyCredential's
5802 [[DiscoverFromExternalSource]](options) method
5803 * 5.7.1. Client data used in WebAuthn signatures (dictionary
5804 CollectedClientData)
5805 * 7.1. Registering a new credential
5806 * 7.2. Verifying an authentication assertion
5807 * 9.4. Client extension processing
5808
5809 #dom-collectedclientdata-authenticatorextensionsReferenced in:
5810 * 5.1.3. Create a new credential - PublicKeyCredential's
5811 [[Create]](options) method
5812

```

```

5552 * 4.1.4. Use an existing credential to make an assertion -
5553 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5554 method
5555 * 4.7.1. Client data used in WebAuthn signatures (dictionary
5556 CollectedClientData)
5557 * 6.1. Registering a new credential
5558 * 6.2. Verifying an authentication assertion
5559
5560 #collectedclientdata-json-serialized-client-dataReferenced in:
5561 * 4.1.3. Create a new credential - PublicKeyCredential's
5562 [[Create]](options) method
5563 * 4.1.4. Use an existing credential to make an assertion -
5564 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5565 method
5566 * 4.2. Authenticator Responses (interface AuthenticatorResponse)
5567 * 4.2.1. Information about Public Key Credential (interface
5568 AuthenticatorAttestationResponse) (2)
5569 * 4.2.2. Web Authentication Assertion (interface
5570 AuthenticatorAssertionResponse)
5571 * 4.7.1. Client data used in WebAuthn signatures (dictionary
5572 CollectedClientData)
5573
5574 #collectedclientdata-hash-of-the-serialized-client-dataReferenced in:
5575 * 4.1.3. Create a new credential - PublicKeyCredential's
5576 [[Create]](options) method (2)
5577 * 4.1.4. Use an existing credential to make an assertion -
5578 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5579 method (2)
5580 * 4.2.1. Information about Public Key Credential (interface
5581 AuthenticatorAttestationResponse)
5582 * 4.2.2. Web Authentication Assertion (interface
5583 AuthenticatorAssertionResponse)
5584 * 4.7.1. Client data used in WebAuthn signatures (dictionary
5585 CollectedClientData)
5586 * 5. WebAuthn Authenticator model
5587 * 5.2.1. The authenticatorMakeCredential operation (2)
5588 * 5.2.2. The authenticatorGetAssertion operation (2) (3)
5589 * 5.3.2. Attestation Statement Formats (2)
5590 * 5.3.4. Generating an Attestation Object
5591 * 6.1. Registering a new credential
5592 * 7.2. Packed Attestation Statement Format (2)
5593 * 7.3. TPM Attestation Statement Format (2)
5594 * 7.4. Android Key Attestation Statement Format (2)
5595 * 7.5. Android SafetyNet Attestation Statement Format
5596 * 7.6. FIDO U2F Attestation Statement Format (2)
5597
5598 #enumdef-publickeycredentialtypeReferenced in:
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-transportReferenced in:
5621 * 4.1.3. Create a new credential - PublicKeyCredential's

```

```

5813 * 5.1.4.1. PublicKeyCredential's
5814 [[DiscoverFromExternalSource]](options) method
5815 * 5.7.1. Client data used in WebAuthn signatures (dictionary
5816 CollectedClientData)
5817 * 7.1. Registering a new credential
5818 * 7.2. Verifying an authentication assertion
5819
5820 #collectedclientdata-json-serialized-client-dataReferenced in:
5821 * 5.1.3. Create a new credential - PublicKeyCredential's
5822 [[Create]](options) method
5823 * 5.1.4.1. PublicKeyCredential's
5824 [[DiscoverFromExternalSource]](options) method
5825 * 5.2. Authenticator Responses (interface AuthenticatorResponse)
5826 * 5.2.1. Information about Public Key Credential (interface
5827 AuthenticatorAttestationResponse) (2)
5828 * 5.2.2. Web Authentication Assertion (interface
5829 AuthenticatorAssertionResponse)
5830 * 5.7.1. Client data used in WebAuthn signatures (dictionary
5831 CollectedClientData)
5832
5833 #collectedclientdata-hash-of-the-serialized-client-dataReferenced in:
5834 * 5.1.3. Create a new credential - PublicKeyCredential's
5835 [[Create]](options) method (2)
5836 * 5.1.4.1. PublicKeyCredential's
5837 [[DiscoverFromExternalSource]](options) method (2)
5838 * 5.2.1. Information about Public Key Credential (interface
5839 AuthenticatorAttestationResponse)
5840 * 5.2.2. Web Authentication Assertion (interface
5841 AuthenticatorAssertionResponse)
5842 * 5.7.1. Client data used in WebAuthn signatures (dictionary
5843 CollectedClientData)
5844 * 6. WebAuthn Authenticator model
5845 * 6.2.1. The authenticatorMakeCredential operation
5846 * 6.2.2. The authenticatorGetAssertion operation (2) (3)
5847 * 6.3.2. Attestation Statement Formats (2)
5848 * 6.3.4. Generating an Attestation Object
5849 * 7.1. Registering a new credential
5850 * 8.2. Packed Attestation Statement Format (2)
5851 * 8.3. TPM Attestation Statement Format (2)
5852 * 8.4. Android Key Attestation Statement Format (2)
5853 * 8.5. Android SafetyNet Attestation Statement Format
5854 * 8.6. FIDO U2F Attestation Statement Format (2)
5855
5856 #enumdef-publickeycredentialtypeReferenced in:
5857 * 5.1.3. Create a new credential - PublicKeyCredential's
5858 [[Create]](options) method (2)
5859 * 5.3. Parameters for Credential Generation (dictionary
5860 PublicKeyCredentialParameters)
5861 * 5.7.2. Credential Type enumeration (enum PublicKeyCredentialType)
5862 * 5.7.3. Credential Descriptor (dictionary
5863 PublicKeyCredentialDescriptor)
5864 * 6.2.1. The authenticatorMakeCredential operation (2) (3)
5865
5866 #dom-publickeycredentialtype-public-keyReferenced in:
5867 * 5.7.2. Credential Type enumeration (enum PublicKeyCredentialType)
5868
5869 #dictdef-publickeycredentialdescriptorReferenced in:
5870 * 5.4. Options for Credential Creation (dictionary
5871 MakePublicKeyCredentialOptions) (2)
5872 * 5.5. Options for Assertion Generation (dictionary
5873 PublicKeyCredentialRequestOptions) (2) (3)
5874 * 5.7.3. Credential Descriptor (dictionary
5875 PublicKeyCredentialDescriptor)
5876 * 6.2.1. The authenticatorMakeCredential operation
5877
5878 #dom-publickeycredentialdescriptor-transportReferenced in:
5879 * 5.1.3. 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
5662 * 4.3. Parameters for Credential Generation (dictionary
5663 PublicKeyCredentialParameters)
5664 * 4.7.5. Cryptographic Algorithm Identifier (typedef
5665 COSEAlgorithmIdentifier)
5666 * 5.2.1. The authenticatorMakeCredential operation
5667 * 5.3.1. Attestation data
5668
5669 #attestation-signatureReferenced in:
5670 * 3. Terminology
5671 * 5. WebAuthn Authenticator model (2) (3)
5672 * 5.3. Attestation
5673 * 7.6. FIDO U2F Attestation Statement Format
5674
5675 #assertion-signatureReferenced in:
5676 * 5. WebAuthn Authenticator model (2)
5677 * 5.2.2. The authenticatorGetAssertion operation (2) (3) (4) (5) (6)
5678
5679 #authenticator-dataReferenced in:
5680 * 4.2.1. Information about Public Key Credential (interface
5681 AuthenticatorAttestationResponse) (2)
5682 * 4.2.2. Web Authentication Assertion (interface
5683 AuthenticatorAssertionResponse)
5684 * 5. WebAuthn Authenticator model (2)
5685 * 5.1. Authenticator data (2) (3) (4) (5) (6) (7) (8)
5686 * 5.2.1. The authenticatorMakeCredential operation (2)
5687 * 5.2.2. The authenticatorGetAssertion operation (2) (3) (4)
5688 * 5.3. Attestation (2)

```

```

5880 [[Create]](options) method (2)
5881 * 5.1.4.1. PublicKeyCredential's
5882 [[DiscoverFromExternalSource]](options) method (2)
5883
5884 #dom-publickeycredentialdescriptor-typeReferenced in:
5885 * 5.1.4.1. PublicKeyCredential's
5886 [[DiscoverFromExternalSource]](options) method
5887 * 5.7.3. Credential Descriptor (dictionary
5888 PublicKeyCredentialDescriptor)
5889
5890 #dom-publickeycredentialdescriptor-idReferenced in:
5891 * 5.1.4.1. PublicKeyCredential's
5892 [[DiscoverFromExternalSource]](options) method
5893 * 5.7.3. Credential Descriptor (dictionary
5894 PublicKeyCredentialDescriptor)
5895
5896 #enumdef-authenticatortransportReferenced in:
5897 * 5.7.3. Credential Descriptor (dictionary
5898 PublicKeyCredentialDescriptor)
5899 * 5.7.4. Authenticator Transport enumeration (enum
5900 AuthenticatorTransport)
5901
5902 #dom-authenticatortransport-usbReferenced in:
5903 * 5.7.4. Authenticator Transport enumeration (enum
5904 AuthenticatorTransport)
5905
5906 #dom-authenticatortransport-nfcReferenced in:
5907 * 5.7.4. Authenticator Transport enumeration (enum
5908 AuthenticatorTransport)
5909
5910 #dom-authenticatortransport-bleReferenced in:
5911 * 5.7.4. Authenticator Transport enumeration (enum
5912 AuthenticatorTransport)
5913
5914 #typedefdef-cosealgorithmidentifierReferenced in:
5915 * 5.1.3. Create a new credential - PublicKeyCredential's
5916 [[Create]](options) method
5917 * 5.3. Parameters for Credential Generation (dictionary
5918 PublicKeyCredentialParameters)
5919 * 5.7.5. Cryptographic Algorithm Identifier (typedef
5920 COSEAlgorithmIdentifier)
5921 * 6.2.1. The authenticatorMakeCredential operation
5922 * 6.3.1. Attestation data
5923 * 8.2. Packed Attestation Statement Format
5924 * 8.3. TPM Attestation Statement Format
5925
5926 #attestation-signatureReferenced in:
5927 * 4. Terminology
5928 * 6. WebAuthn Authenticator model (2) (3)
5929 * 6.3. Attestation
5930 * 8.6. FIDO U2F Attestation Statement Format
5931
5932 #assertion-signatureReferenced in:
5933 * 6. WebAuthn Authenticator model (2)
5934 * 6.2.2. The authenticatorGetAssertion operation (2) (3) (4) (5) (6)
5935 (7)
5936
5937 #authenticator-dataReferenced in:
5938 * 5.2.1. Information about Public Key Credential (interface
5939 AuthenticatorAttestationResponse) (2)
5940 * 5.2.2. Web Authentication Assertion (interface
5941 AuthenticatorAssertionResponse)
5942 * 6. WebAuthn Authenticator model (2)
5943 * 6.1. Authenticator data (2) (3) (4) (5) (6) (7) (8)
5944 * 6.1.1. Signature Counter Considerations (2)
5945 * 6.2.1. The authenticatorMakeCredential operation
5946 * 6.2.2. The authenticatorGetAssertion operation (2) (3) (4)

```

- 5685 \* 5.3.1. Attestation data
- 5690 \* 5.3.2. Attestation Statement Formats (2)
- 5691 \* 5.3.4. Generating an Attestation Object (2) (3)
- 5692 \* 5.3.5.3. Attestation Certificate Hierarchy
- 5693 \* 6.1. Registering a new credential (2)
- 5694 \* 7.5. Android SafetyNet Attestation Statement Format
- 5695 \* 8.5. Authenticator extension processing (2)
- 5696 \* 8.6. Example Extension (2)
- 5697 \* 9.6. User Verification Index Extension (uvi)
- 5698 \* 9.7. Location Extension (loc)
- 5699 \* 9.8. User Verification Method Extension (uvm)

- 5700
- 5701 #authenticatormakecredentialReferenced in:
- 5702 \* 3. Terminology (2) (3)
- 5703 \* 4.1.3. Create a new credential - PublicKeyCredential's
- 5704 [[Create]](options) method (2)
- 5705 \* 5. WebAuthn Authenticator model
- 5706 \* 5.2.3. The authenticatorCancel operation (2)
- 5707 \* 8. WebAuthn Extensions
- 5708 \* 8.2. Defining extensions
- 5709
- 5710 #authenticatorgetassertionReferenced in:
- 5711 \* 3. Terminology (2) (3)
- 5712 \* 4.1.4. Use an existing credential to make an assertion -
- 5713 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
- 5714 method (2) (3) (4)
- 5715 \* 5. WebAuthn Authenticator model
- 5716 \* 5.1. Authenticator data
- 5717 \* 5.2.3. The authenticatorCancel operation (2)
- 5718 \* 8. WebAuthn Extensions
- 5719 \* 8.2. Defining extensions
- 5720
- 5721 #authenticatorcancelReferenced in:
- 5722 \* 4.1.3. Create a new credential - PublicKeyCredential's
- 5723 [[Create]](options) method (2) (3)
- 5724 \* 4.1.4. Use an existing credential to make an assertion -
- 5725 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
- 5726 method (2) (3)
- 5727
- 5728 #attestation-objectReferenced in:
- 5729 \* 3. Terminology
- 5730 \* 4. Web Authentication API
- 5731 \* 4.2.1. Information about Public Key Credential (interface
- 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:

- 5947 \* 6.3. Attestation (2)
- 5948 \* 6.3.1. Attestation data
- 5949 \* 6.3.2. Attestation Statement Formats (2)
- 5950 \* 6.3.4. Generating an Attestation Object
- 5951 \* 6.3.5.3. Attestation Certificate Hierarchy
- 5952 \* 7.1. Registering a new credential (2)
- 5953 \* 8.5. Android SafetyNet Attestation Statement Format
- 5954 \* 9.5. Authenticator extension processing
- 5955 \* 9.6. Example Extension (2)
- 5956 \* 10.6. User Verification Index Extension (uvi)
- 5957 \* 10.7. Location Extension (loc)
- 5958 \* 10.8. User Verification Method Extension (uvm)
- 5959
- 5960 #signature-counterReferenced in:
- 5961 \* 6.1.1. Signature Counter Considerations (2) (3) (4) (5) (6) (7) (8)
- 5962 (9) (10) (11) (12)
- 5963 \* 6.2.1. The authenticatorMakeCredential operation (2) (3) (4)
- 5964 \* 6.2.2. The authenticatorGetAssertion operation (2)
- 5965 \* 7.1. Registering a new credential
- 5966 \* 7.2. Verifying an authentication assertion (2) (3) (4) (5) (6)
- 5967
- 5968 #authenticatormakecredentialReferenced in:
- 5969 \* 4. Terminology (2) (3)
- 5970 \* 5.1.3. Create a new credential - PublicKeyCredential's
- 5971 [[Create]](options) method (2)
- 5972 \* 6. WebAuthn Authenticator model
- 5973 \* 6.2.3. The authenticatorCancel operation (2)
- 5974 \* 9. WebAuthn Extensions
- 5975 \* 9.2. Defining extensions
- 5976
- 5977 #authenticatorgetassertionReferenced in:
- 5978 \* 4. Terminology (2) (3)
- 5979 \* 5.1.4.1. PublicKeyCredential's
- 5980 [[DiscoverFromExternalSource]](options) method (2) (3) (4) (5)
- 5981 \* 6. WebAuthn Authenticator model
- 5982 \* 6.1. Authenticator data
- 5983 \* 6.1.1. Signature Counter Considerations (2) (3)
- 5984 \* 6.2.3. The authenticatorCancel operation (2)
- 5985 \* 9. WebAuthn Extensions
- 5986 \* 9.2. Defining extensions
- 5987
- 5988 #authenticatorcancelReferenced in:
- 5989 \* 5.1.3. Create a new credential - PublicKeyCredential's
- 5990 [[Create]](options) method (2) (3)
- 5991 \* 5.1.4.1. PublicKeyCredential's
- 5992 [[DiscoverFromExternalSource]](options) method (2) (3)
- 5993
- 5994 #attestation-objectReferenced in:
- 5995 \* 4. Terminology
- 5996 \* 5. Web Authentication API
- 5997 \* 5.2.1. Information about Public Key Credential (interface
- 5998 AuthenticatorAttestationResponse) (2)
- 5999 \* 5.4. Options for Credential Creation (dictionary
- 6000 MakePublicKeyCredentialOptions) (2)
- 6001 \* 6.2.1. The authenticatorMakeCredential operation (2)
- 6002 \* 6.3. Attestation (2) (3)
- 6003 \* 6.3.1. Attestation data
- 6004 \* 6.3.4. Generating an Attestation Object (2)
- 6005 \* 7.1. Registering a new credential
- 6006
- 6007 #attestation-statementReferenced in:
- 6008 \* 4. Terminology
- 6009 \* 5.2.1. Information about Public Key Credential (interface
- 6010 AuthenticatorAttestationResponse) (2) (3)
- 6011 \* 6.3. Attestation (2) (3) (4) (5) (6) (7) (8)
- 6012 \* 6.3.2. Attestation Statement Formats (2) (3)
- 6013 \* 7.1. Registering a new credential
- 6014
- 6015 #attestation-statement-formatReferenced in:

5745 \* 4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

5751 \* 4.7.4. Authenticator Transport enumeration (enum 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 #attestation-typeReferenced in:

5757 \* 5.3. Attestation (2) (3) (4) (5) (6)

5758 \* 5.3.2. Attestation Statement Formats

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

5771 #signing-procedureReferenced in:

5772 \* 5.3.2. Attestation Statement Formats

5773 #authenticator-data-for-the-attestationReferenced in:

5774 \* 7.2. Packed Attestation Statement Format

5775 \* 7.3. TPM Attestation Statement Format

5776 \* 7.4. Android Key Attestation Statement Format (2)

5777 \* 7.5. Android SafetyNet Attestation Statement Format

5778 \* 7.6. FIDO U2F Attestation Statement Format

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 #basic-attestationReferenced in:

5787 \* 5.3.5.1. Privacy

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 Compromise

5797 \* 6.1. Registering a new credential (2) (3)

5798 \* 7.2. Packed Attestation Statement Format (2)

5799 \* 7.6. FIDO U2F Attestation Statement Format

5801 #privacy-caReferenced in:

5802 \* 5.3.5.1. Privacy

5803 #elliptic-curve-based-direct-anonymous-attestationReferenced in:

5804 \* 5.3.5.1. Privacy

5805 #ecdaaReferenced in:

5806 \* 5.3.2. Attestation Statement Formats

5807 \* 5.3.3. Attestation Types

5808 \* 5.3.5.2. Attestation Certificate and Attestation Certificate CA Compromise

5809 \* 6.1. Registering a new credential

5810 \* 7.2. Packed Attestation Statement Format (2)

6016 \* 5.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

6017 \* 5.7.4. Authenticator Transport enumeration (enum AuthenticatorTransport)

6018 \* 6.2.1. The authenticatorMakeCredential operation

6019 \* 6.3. Attestation (2) (3) (4) (5) (6) (7)

6020 \* 6.3.2. Attestation Statement Formats (2) (3) (4)

6021 \* 6.3.4. Generating an Attestation Object

6022 \* 7.1. Registering a new credential

6023 #attestation-typeReferenced in:

6024 \* 6.3. Attestation (2) (3) (4) (5) (6)

6025 \* 6.3.2. Attestation Statement Formats

6026 #attestation-dataReferenced in:

6027 \* 6.1. Authenticator data (2) (3) (4) (5) (6) (7)

6028 \* 6.2.1. The authenticatorMakeCredential operation

6029 \* 6.2.2. The authenticatorGetAssertion operation

6030 \* 6.3. Attestation (2)

6031 \* 6.3.3. Attestation Types

6032 \* 7.1. Registering a new credential

6033 \* 8.3. TPM Attestation Statement Format

6034 \* 8.4. Android Key Attestation Statement Format

6035 \* 8.6. FIDO U2F Attestation Statement Format

6036 #signing-procedureReferenced in:

6037 \* 6.3.2. Attestation Statement Formats

6038 \* 6.3.4. Generating an Attestation Object

6039 #authenticator-data-for-the-attestationReferenced in:

6040 \* 8.2. Packed Attestation Statement Format

6041 \* 8.3. TPM Attestation Statement Format

6042 \* 8.4. Android Key Attestation Statement Format (2)

6043 \* 8.5. Android SafetyNet Attestation Statement Format

6044 \* 8.6. FIDO U2F Attestation Statement Format

6045 #authenticator-data-claimed-to-have-been-used-for-the-attestationReferenced in:

6046 \* 8.2. Packed Attestation Statement Format

6047 \* 8.3. TPM Attestation Statement Format

6048 \* 8.4. Android Key Attestation Statement Format (2)

6049 \* 8.5. Android SafetyNet Attestation Statement Format

6050 \* 8.6. FIDO U2F Attestation Statement Format

6051 #basic-attestationReferenced in:

6052 \* 6.3.5.1. Privacy

6053 #self-attestationReferenced in:

6054 \* 4. Terminology (2) (3) (4)

6055 \* 6.3. Attestation (2)

6056 \* 6.3.2. Attestation Statement Formats

6057 \* 6.3.3. Attestation Types

6058 \* 6.3.5.2. Attestation Certificate and Attestation Certificate CA Compromise

6059 \* 7.1. Registering a new credential (2) (3)

6060 \* 8.2. Packed Attestation Statement Format (2)

6061 \* 8.6. FIDO U2F Attestation Statement Format

6062 #privacy-caReferenced in:

6063 \* 6.3.5.1. Privacy

6064 #elliptic-curve-based-direct-anonymous-attestationReferenced in:

6065 \* 6.3.5.1. Privacy

6066 #ecdaaReferenced in:

6067 \* 6.3.2. Attestation Statement Formats

6068 \* 6.3.3. Attestation Types

6069 \* 6.3.5.2. Attestation Certificate and Attestation Certificate CA Compromise

6070 \* 7.1. Registering a new credential

6071 \* 8.2. 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-eccdaa-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 #eccdaa-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 * 8. WebAuthn Extensions (2) (3) (4) (5) (6)
5852 * 8.6. Example Extension
5853 * 9.1. FIDO Appid Extension (appid)
5854 * 9.2. Simple Transaction Authorization Extension (txAuthSimple)
5855 * 9.3. Generic Transaction Authorization Extension (txAuthGeneric)
5856 * 9.6. User Verification Index Extension (uvi)
5857 * 9.7. Location Extension (loc)
5858 * 9.8. User Verification Method Extension (uvm)
5859 * 10.2. WebAuthn Extension Identifier Registrations (2) (3) (4) (5)
5860 (6)
5861
5862 #client-extensionReferenced in:
5863 * 4.1.3. Create a new credential - PublicKeyCredential's
5864 [[Create]](options) method
5865 * 4.1.4. Use an existing credential to make an assertion -
5866 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5867 method
5868 * 4.6. Authentication Extensions (typedef AuthenticationExtensions)
5869 * 8. WebAuthn Extensions
5870 * 8.2. Defining extensions
5871 * 8.4. Client extension processing
5872
5873 #authenticator-extensionReferenced in:
5874 * 4.1.3. Create a new credential - PublicKeyCredential's
5875 [[Create]](options) method
5876 * 4.1.4. Use an existing credential to make an assertion -
5877 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5878 method
5879 * 4.6. Authentication Extensions (typedef AuthenticationExtensions)
5880 * 8. WebAuthn Extensions (2) (3)
5881 * 8.2. Defining extensions (2)
5882 * 8.3. Extending request parameters
5883 * 8.5. Authenticator extension processing
5884

```

```

6086 * 8.3. TPM Attestation Statement Format (2)
6087
6088 #attestation-statement-format-identifierReferenced in:
6089 * 6.3.2. Attestation Statement Formats
6090 * 6.3.4. Generating an Attestation Object
6091
6092 #identifier-of-the-eccdaa-issuer-public-keyReferenced in:
6093 * 7.1. Registering a new credential
6094 * 8.2. Packed Attestation Statement Format
6095 * 8.3. TPM Attestation Statement Format (2)
6096
6097 #eccdaa-issuer-public-keyReferenced in:
6098 * 6.3.2. Attestation Statement Formats
6099 * 6.3.5.1. Privacy
6100 * 7.1. Registering a new credential
6101 * 8.2. Packed Attestation Statement Format (2) (3)
6102
6103 #registration-extensionReferenced in:
6104 * 5.1.3. Create a new credential - PublicKeyCredential's
6105 [[Create]](options) method
6106 * 9. WebAuthn Extensions (2) (3) (4) (5) (6)
6107 * 9.6. Example Extension
6108 * 10.2. Simple Transaction Authorization Extension (txAuthSimple)
6109 * 10.3. Generic Transaction Authorization Extension (txAuthGeneric)
6110 * 10.4. Authenticator Selection Extension (authnSel)
6111 * 10.5. Supported Extensions Extension (exts)
6112 * 10.6. User Verification Index Extension (uvi)
6113 * 10.7. Location Extension (loc)
6114 * 10.8. User Verification Method Extension (uvm)
6115 * 11.2. WebAuthn Extension Identifier Registrations (2) (3) (4) (5)
6116 (6) (7)
6117
6118 #authentication-extensionReferenced in:
6119 * 5.1.4.1. PublicKeyCredential's
6120 [[DiscoverFromExternalSource]](options) method
6121 * 9. WebAuthn Extensions (2) (3) (4) (5) (6)
6122 * 9.6. Example Extension
6123 * 10.1. FIDO Appid Extension (appid)
6124 * 10.2. Simple Transaction Authorization Extension (txAuthSimple)
6125 * 10.3. Generic Transaction Authorization Extension (txAuthGeneric)
6126 * 10.6. User Verification Index Extension (uvi)
6127 * 10.7. Location Extension (loc)
6128 * 10.8. User Verification Method Extension (uvm)
6129 * 11.2. WebAuthn Extension Identifier Registrations (2) (3) (4) (5)
6130 (6)
6131
6132 #client-extensionReferenced in:
6133 * 5.1.3. Create a new credential - PublicKeyCredential's
6134 [[Create]](options) method
6135 * 5.1.4.1. PublicKeyCredential's
6136 [[DiscoverFromExternalSource]](options) method
6137 * 5.6. Authentication Extensions (typedef AuthenticationExtensions)
6138 * 9. WebAuthn Extensions
6139 * 9.2. Defining extensions
6140 * 9.4. Client extension processing
6141
6142 #authenticator-extensionReferenced in:
6143 * 5.1.3. Create a new credential - PublicKeyCredential's
6144 [[Create]](options) method
6145 * 5.1.4.1. PublicKeyCredential's
6146 [[DiscoverFromExternalSource]](options) method
6147 * 5.6. Authentication Extensions (typedef AuthenticationExtensions)
6148 * 9. WebAuthn Extensions (2) (3)
6149 * 9.2. Defining extensions (2)
6150 * 9.3. Extending request parameters
6151 * 9.5. Authenticator extension processing
6152

```

```

5885 #extension-identifierReferenced in:
5886 * 4.1. PublicKeyCredential Interface
5887 * 4.1.3. Create a new credential - PublicKeyCredential's
5888 [[Create]](options) method
5889 * 4.1.4. Use an existing credential to make an assertion -
5890 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5891 method
5892 * 5.1. Authenticator data
5893 * 8. WebAuthn Extensions (2)
5894 * 8.2. Defining extensions
5895 * 8.3. Extending request parameters
5896 * 8.4. Client extension processing (2)
5897 * 8.5. Authenticator extension processing (2)
5898 * 8.6. Example Extension
5899 * 9.5. Supported Extensions Extension (exts) (2)
5900 * 9.7. Location Extension (loc)
5901 * 10.2. WebAuthn Extension Identifier Registrations
5902
5903 #client-extension-inputReferenced in:
5904 * 8. WebAuthn Extensions (2) (3)
5905 * 8.2. Defining extensions
5906 * 8.3. Extending request parameters (2) (3) (4) (5) (6)
5907 * 8.4. Client extension processing (2) (3) (4)
5908 * 8.6. Example Extension
5909
5910 #authenticator-extension-inputReferenced in:
5911 * 8. WebAuthn Extensions (2) (3) (4) (5)
5912 * 8.2. Defining extensions
5913 * 8.3. Extending request parameters (2) (3)
5914 * 8.4. Client extension processing
5915 * 8.5. Authenticator extension processing (2) (3)
5916
5917 #client-extension-processingReferenced in:
5918 * 4.1. PublicKeyCredential Interface
5919 * 4.1.3. Create a new credential - PublicKeyCredential's
5920 [[Create]](options) method (2)
5921 * 4.1.4. Use an existing credential to make an assertion -
5922 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5923 method (2)
5924 * 8. WebAuthn Extensions (2) (3) (4)
5925 * 8.2. Defining extensions
5926
5927 #client-extension-outputReferenced in:
5928 * 4.1. PublicKeyCredential Interface
5929 * 4.1.3. Create a new credential - PublicKeyCredential's
5930 [[Create]](options) method (2)
5931 * 4.1.4. Use an existing credential to make an assertion -
5932 PublicKeyCredential's [[DiscoverFromExternalSource]](options)
5933 method (2)
5934 * 8. WebAuthn Extensions (2) (3)
5935 * 8.2. Defining extensions (2) (3)
5936 * 8.4. Client extension processing (2) (3)
5937 * 8.6. Example Extension
5938
5939 #authenticator-extension-processingReferenced in:
5940 * 8. WebAuthn Extensions
5941 * 8.2. Defining extensions
5942 * 8.5. Authenticator extension processing
5943
5944 #authenticator-extension-outputReferenced in:
5945 * 5.1. Authenticator data
5946 * 8. WebAuthn Extensions (2) (3)
5947 * 8.2. Defining extensions (2) (3)
5948 * 8.4. Client extension processing
5949 * 8.5. Authenticator extension processing
5950 * 8.6. Example Extension
5951 * 9.5. Supported Extensions Extension (exts)
5952 * 9.6. User Verification Index Extension (uvi)

```

```

6153 #extension-identifierReferenced in:
6154 * 5.1. PublicKeyCredential Interface
6155 * 5.1.3. Create a new credential - PublicKeyCredential's
6156 [[Create]](options) method
6157 * 5.1.4.1. PublicKeyCredential's
6158 [[DiscoverFromExternalSource]](options) method
6159 * 6.1. Authenticator data
6160 * 6.2.1. The authenticatorMakeCredential operation
6161 * 9. WebAuthn Extensions (2)
6162 * 9.2. Defining extensions
6163 * 9.3. Extending request parameters
6164 * 9.4. Client extension processing (2)
6165 * 9.5. Authenticator extension processing (2)
6166 * 9.6. Example Extension
6167 * 10.5. Supported Extensions Extension (exts) (2)
6168 * 10.7. Location Extension (loc)
6169 * 11.2. WebAuthn Extension Identifier Registrations
6170
6171 #client-extension-inputReferenced in:
6172 * 9. WebAuthn Extensions (2) (3)
6173 * 9.2. Defining extensions
6174 * 9.3. Extending request parameters (2) (3) (4) (5) (6)
6175 * 9.4. Client extension processing (2) (3) (4)
6176 * 9.6. Example Extension
6177
6178 #authenticator-extension-inputReferenced in:
6179 * 6.2.1. The authenticatorMakeCredential operation
6180 * 9. WebAuthn Extensions (2) (3) (4) (5)
6181 * 9.2. Defining extensions
6182 * 9.3. Extending request parameters (2) (3)
6183 * 9.4. Client extension processing
6184 * 9.5. Authenticator extension processing (2) (3)
6185
6186 #client-extension-processingReferenced in:
6187 * 5.1. PublicKeyCredential Interface
6188 * 5.1.3. Create a new credential - PublicKeyCredential's
6189 [[Create]](options) method (2)
6190 * 5.1.4.1. PublicKeyCredential's
6191 [[DiscoverFromExternalSource]](options) method (2)
6192 * 9. WebAuthn Extensions (2) (3) (4)
6193 * 9.2. Defining extensions
6194
6195 #client-extension-outputReferenced in:
6196 * 5.1. PublicKeyCredential Interface
6197 * 5.1.3. Create a new credential - PublicKeyCredential's
6198 [[Create]](options) method (2)
6199 * 5.1.4.1. PublicKeyCredential's
6200 [[DiscoverFromExternalSource]](options) method (2)
6201 * 9. WebAuthn Extensions (2) (3)
6202 * 9.2. Defining extensions (2) (3)
6203 * 9.4. Client extension processing (2) (3)
6204 * 9.6. Example Extension
6205
6206 #authenticator-extension-processingReferenced in:
6207 * 6.2.1. The authenticatorMakeCredential operation
6208 * 9. WebAuthn Extensions
6209 * 9.2. Defining extensions
6210 * 9.5. Authenticator extension processing
6211
6212 #authenticator-extension-outputReferenced in:
6213 * 6.1. Authenticator data
6214 * 9. WebAuthn Extensions (2) (3)
6215 * 9.2. Defining extensions (2) (3)
6216 * 9.4. Client extension processing
6217 * 9.5. Authenticator extension processing
6218 * 9.6. Example Extension
6219 * 10.5. Supported Extensions Extension (exts)
6220 * 10.6. 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

6221 \* 10.7. Location Extension (loc)  
6222 \* 10.8. User Verification Method Extension (uvm)  
6223  
6224 #typedefdef-authenticatorselectionlistReferenced in:  
6225 \* 10.4. Authenticator Selection Extension (authnSel)  
6226  
6227 #typedefdef-aaguidReferenced in:  
6228 \* 10.4. Authenticator Selection Extension (authnSel)  
6229