This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical
1.1.2. Authentication

This section is not normative.

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

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

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

1.1. Use Cases

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

1.1.1. Registration

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

1.1.2. Authentication

This section is not normative.

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

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

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

1.1. Use Cases

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

1.1.1. Registration

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

Compact Binary Object Representation (CBOR) [RFC7049]. 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 Relying Party prompts the user for their authorization gesture in order to authorize a single transaction, such as a payment or other financial transaction.

1.2. Dependencies

This specification defines criteria for a Conforming User Agent: A User Agent MUST behave as described in this specification in any way desired, so long as the 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-2].

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

2.1. Dependencies

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

Base64url encoding

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

CBOR

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

CDDL

This specification describes the syntax of all CBOR-encoded data using the CBOR Data Definition Language (CDDL) [CDDL].
The ceremony where a user, and the user's computing device(s) (containing at least one authenticator) work in concert to cryptographically prove to a Relying Party that the user controls the credential private key associated with a previously-registered authenticator.
This specification, and handling communication between
platform, the Web Authentication API and algorithms given in
concept of a network protocol, with human nodes alongside
Registration and Authentication are ceremonies, and an
authorization gesture is often a component of those ceremonies.

Client
See Conforming User Agent.

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

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

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

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

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

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

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

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

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

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

Rate Limiting

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

Registration

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

Relying Party Identifier

RP ID

A valid domain string that identifies the Relying Party on whose behalf a given registration or authentication ceremony is being performed. A public key credential can only be used for authentication by the same entity (as identified by RP ID) that created and registered them. By default, the RP ID for a WebAuthn operation is set to the caller’s origin’s effective domain. This default MAY be overridden by the caller, subject to certain restrictions, as discussed further in 4.1.3 Create a new credential - PublicKeyCredential’s [[Create]](options) method and 4.1.4 Use existing credential - PublicKeyCredential::[[DiscoverFromExternalSource]](options) method.

Credentialed Public Key

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

Public Key Credential

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

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

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

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

Credential Public Key

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

Registration

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

Relying Party

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

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

Relying Party Identifier

RP ID

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

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

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

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

Public Key Credential

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

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

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

This is done in order to match the behavior of pervasively deployed ambient credentials (e.g., cookies, [RFC6265]). Please note that this is a greater relaxation of "same-origin" restrictions than what document.domain's setter provides.
Generically, a credential is data one entity presents to another in order to authenticate the former to the latter [RFC4949]. A WebAuthn public key credential is a \(\langle\text{identifier, type}\rangle\) pair identifying authentication information established by the authenticator and the Relying Party, together, at registration time. The authentication information consists of an asymmetric key pair, where the public key portion is returned to the Relying Party, which stores it in conjunction with the present user's account. The authenticator maps the private key to the Relying Party’s RP ID and stores it. Subsequently, only that Relying Party, as identified by its RP ID, is able to employ the public key credential in authentication ceremonies, via the \texttt{get()} method. The Relying Party uses its copy of the stored public key to verify the resultant Authentication Assertion.

Test of User Presence

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

Client-side-resident Credential Private Key

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

Client-Side

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

User Consent

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

User Verification

The technical process by which an authenticator locally authorizes the invocation of the authenticator\texttt{MakeCredential} and \texttt{GetAssertion} operations implies use of key material managed by the authenticator. Note that for security, user verification and use of credential private keys must occur within a single logical security boundary defining the authenticator.

Test of User Presence

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

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

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

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

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

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

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

4.1. PublicKeyCredential Interface

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

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

4.1. PublicKeyCredential Interface

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

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

```javascript
[SecureContext]
interface PublicKeyCredential : Credential {
  
  
  [SameObject] readonly attribute ArrayBuffer rawId;
  
  
  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 AuthenticatorAssertionResponse; otherwise, the
  PublicKeyCredential was created in response to get(), and this
  attribute's value will be an AuthenticatorAssertionResponse.

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

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

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

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

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

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

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

  partial dictionary CredentialRequestOptions {
  ```
3. If options.rp.id is not a registrable domain suffix of and is
2. If effectiveDomain is null, then return a DOMException whose
1. Let effectiveDomain be the callerOrigin's effective domain.

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

1. Assert: options.publicKey is present.
2. Let options be the value of options.publicKey.
3. If any of the name member of options.rp, the name member of
   options.user, the displayName member of options.user, or the id
   member of options.user are not present, return a TypeError simple
   exception.
4. If the timeout member of options is present, check if its value
   lies within a reasonable range as defined by the platform and if
   not, correct it to the closest value lying within that range. Set
   adjustedTimeout to this adjusted value. If the timeout member of
   options is not present, then set adjustedTimeout to a
   platform-specific default.
5. Let global be the PublicKeyCredential's interface object's
   environment settings object's global object.
6. Let callerOrigin be the origin specified by this
   PublicKeyCredential's interface object's relevant settings object.
   If callerOrigin is an opaque origin, return a DOMException whose
   name is "NotAllowedError", and terminate this algorithm.
7. If the id member of options.rp is not present, then set rpId to
callerOrigin.
   Otherwise:
   1. Let effectiveDomain be the callerOrigin's effective domain.
   2. If effectiveDomain is null, then return a DOMException whose
      name is "SecurityError" and terminate this algorithm.
   3. If options.rp.id is not a registrable domain suffix of and is

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

partial dictionary CredentialCreationOptions {
  PublicKeyCredentialRequestOptions publicKey;
}

4.1.2. CredentialRequestOptions Extension

Part of this specification is taken care of by navigator.credentials.create().

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

This method accepts a single argument:

options

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

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

1. Assert: options.publicKey is present.
2. Let options be the value of options.publicKey.
3. If any of the name member of options.rp, the name member of
   options.user, the displayName member of options.user, or the id
   member of options.user are not present, return a TypeError simple
   exception.
4. If the timeout member of options is present, check if its value
   lies within a reasonable range as defined by the platform and if
   not, correct it to the closest value lying within that range. Set
   adjustedTimeout to this adjusted value. If the timeout member of
   options is not present, then set adjustedTimeout to a
   platform-specific default.
5. Let global be the PublicKeyCredential's interface object's
   environment settings object's global object.
6. Let callerOrigin be the origin specified by this
   PublicKeyCredential's interface object's relevant settings object.
   If callerOrigin is an opaque origin, return a DOMException whose
   name is "NotAllowedError", and terminate this algorithm.
7. If the id member of options.rp is not present, then set rpId to
callerOrigin.
   Otherwise:
   1. Let effectiveDomain be the callerOrigin's effective domain.
   2. If effectiveDomain is null, then return a DOMException whose
      name is "SecurityError" and terminate this algorithm.
   3. If options.rp.id is not a registrable domain suffix of and is

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

partial dictionary CredentialRequestOptions {
  PublicKeyCredentialRequestOptions publicKey;
}

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

Part of this specification is handled by navigator.credentials.create().

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

This method accepts a single argument:

options

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

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

1. Assert: options.publicKey is present.
2. Let options be the value of options.publicKey.
3. If any of the name member of options.rp, the name member of
   options.user, the displayName member of options.user, or the id
   member of options.user are not present, return a TypeError simple
   exception.
4. If the timeout member of options is present, check if its value
   lies within a reasonable range as defined by the platform and if
   not, correct it to the closest value lying within that range. Set
   adjustedTimeout to this adjusted value. If the timeout member of
   options is not present, then set adjustedTimeout to a
   platform-specific default.
5. Let global be the PublicKeyCredential's interface object's
   environment settings object's global object.
6. Let callerOrigin be the origin specified by this
   PublicKeyCredential's interface object's relevant settings object.
   If callerOrigin is an opaque origin, return a DOMException whose
   name is "NotAllowedError", and terminate this algorithm.
7. If the id member of options.rp is not present, then set rpId to
callerOrigin.
   Otherwise:
   1. Let effectiveDomain be the callerOrigin's effective domain.
   2. If effectiveDomain is null, then return a DOMException whose
      name is "SecurityError" and terminate this algorithm.
   3. If options.rp.id is not a registrable domain suffix of and is


18. If currentlyAvailableAuthenticators is empty, return a DOMException whose name is "SecurityError", and terminate this algorithm.

17. Let selectedAuthenticators be a new ordered set consisting of all authenticators available on this platform.

16. Let currentlyAvailableAuthenticators be a new ordered set represented by clientDataJSON.

15. Let clientDataHash be the hash of the serialized client data.

14. For each current of options.parameters:
   1. If current.type does not contain a PublicKeyCredentialType supported by this implementation, then continue.
   2. Let normalizedAlgorithm be the result of normalizing an algorithm [WebCryptoAPI], with alg set to current.alg.
   3. If normalizedAlgorithm is not an authenticator extension, then continue.
   4. Set authenticatorExtensionInput to the (CBOR) result of running extensionId's client extension processing algorithm on clientExtensionsInput. If the algorithm returned an error, continue.

13. Let collectedClientData be a new CollectedClientData instance whose fields are:
   1. challenge
   2. origin
   3. hashAlg
   4. tokenBinding
   5. clientExtensions
   6. authenticatorExtensions

12. If the extensions member of options is present, then for each extensionId -> clientExtensionInput of options.extensions:
   1. If extensionId is not supported by this client platform or is not a registration extension, then continue.
   2. Set clientExtensions[extensionId] to clientExtensionInput.
   3. If extensionId is not an authenticator extension, then continue.

11. For each current of options.pubKeyCredParams:
   1. If current.type does not contain a PublicKeyCredentialType supported by this platform or is not a registration extension, then continue.
   2. Set rpId to options.rp.id.

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

9. For each current of options.pubKeyCredParams:
   1. If current.type does not contain a PublicKeyCredentialType supported by this implementation, then continue.
   2. Let alg be current.alg.

8. Set rpid to options.rp.id.

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

6. Let clientDataHash be the hash of the caller's RP ID. The RP ID defaults to being the caller's origin's effective domain unless the caller has explicitly set options.rp.id when calling create().

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

4. Let selectedAuthenticators be a new ordered set consisting of all authenticators available on this platform.

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

2. Set rpId to current.rpId.

1. Set authenticatorExtensions to being a new map and let authenticatorExtensions be a new map.

18. If currentlyAvailableAuthenticators is empty, return a DOMException whose name is "SecurityError", and terminate this algorithm.

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

16. Let collectedClientData be a new CollectedClientData instance whose fields are:
   1. challenge
   2. origin
   3. hashAlg
   4. tokenBinding
   5. clientExtensions
   6. authenticatorExtensions

15. Let collectedClientData be a new CollectedClientData instance whose fields are:
   1. challenge
   2. origin
   3. hashAlg
   4. tokenBinding
   5. clientExtensions
   6. authenticatorExtensions

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

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

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

11. Let selectedAuthenticators be a new ordered set.

10. If currentlyAvailableAuthenticators is empty, return a DOMException
whose name is "NotFoundError", and terminate this algorithm.

21. If options.authenticatorSelection is present, iterate through currentlyAvailableAuthenticators and do the following for each authenticator:
   1. If attachment is present and its value is not equal to authenticator's attachment modality, continue.
   2. If requireResidentKey is set to true and the authenticator is not capable of storing a Client-Side-Resident Credential Private Key, continue.
   3. Append authenticator to selectedAuthenticators.

22. If selectedAuthenticators is empty, return a DOMException whose name is "ConstraintError", and terminate this algorithm.

23. Let issuedRequests be a new ordered set.

24. For each authenticator in currentlyAvailableAuthenticators:
   1. Let excludeList be a new list.
   2. For each credential C in options.excludeCredentialDescriptorList:
      1. If C.transports is not empty, and authenticator is connected over a transport not mentioned in C.transports, the client MAY continue.
      2. Otherwise, Append C to excludeList.
   3. In parallel, invoke the authenticatorMakeCredential operation on authenticator with options.user, normalizedParameters, excludeList, and authenticatorExtensions as parameters.

4. Append authenticator to issuedRequests.

25. Start a timer for adjustedTimeout milliseconds. Then execute the following steps in parallel. The task source for these tasks is the manipulation task source.

26. While issuedRequests is not empty, perform the following actions depending upon the adjustedTimeout timer and responses from the authenticators:

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

   If any authenticator returns a status indicating that the user cancelled the operation,
   1. Remove authenticator from issuedRequests.
   2. For each remaining authenticator in issuedRequests invoke the authenticatorCancel operation on authenticator and remove it from issuedRequests.

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

   If any authenticator indicates success,
   1. Remove authenticator from issuedRequests.
   2. Let attestationObject be a new ArrayBuffer, created using global's %ArrayBuffer%, containing the bytes of the value returned from the successful authenticatorMakeCredential operation (which is attObj, as defined in 5.3.4 Generating an Attestation Object).
   3. Let id be attestationObject.authData.attestation data.credential ID (see 5.3.1 Attestation data and 5.1 Authenticator data).
   4. Let value be a new PublicKeyCredential object associated with global whose fields are:
      [[identifier]]
      id
      response
4. Let callerOrigin be the origin of this CredentialsContainer.

3. Let global be the PublicKeyCredential’s relevant settings object’s environment settings object’s global object.

2. If the timeout member of publicKeyOptions is present, check if its value lies within a reasonable range as defined by the platform and if not, correct it to the closest value lying within that range.

1. Let publicKeyOptions be the value of options publicKey member.

algorithm:

options:

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

A PublicKeyCredentialRequestOptions

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

1. Let publicKeyOptions be the value of options publicKey member.

2. If the timeout member of publicKeyOptions is present, check if its value lies within a reasonable range as defined by the platform and if not, correct it to the closest value lying within that range.

3. Global be the PublicKeyCredential’s relevant settings object’s environment settings object’s global object.

4. Let callerOrigin be the origin of this CredentialsContainer.

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

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

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

This method takes the following parameters:

options:

This argument is a CredentialRequestOptions object whose publicKey property optionally specifies additional options as described in 4.5 Options for Assertion Generation (dictionary).

The selected authenticator signs the challenge along with other collected data in order to produce an assertion. See 5.2.2 The AuthenticatorGetAssertion operation.

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

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

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

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

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

This method accepts a single argument:

options:

This argument is a CredentialRequestOptions object whose publicKey property optionally specifies additional options as described in 4.5 Options for Assertion Generation (dictionary).

The selected authenticator signs the challenge along with other collected data in order to produce an assertion. See 5.2.2 The AuthenticatorGetAssertion operation.

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

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

4. Let callerOrigin be the origin specified by this

5. Let callerOrigin be the origin of this

6. Return value and terminate this algorithm.

5. For each remaining authenticator in issuedRequests invoke

the authentictCancel operation on authenticator and remove it from issuedRequests.

6. Return value and terminate this algorithm.
13. For each authenticator currently available on this platform, return a DOMException whose name is "NotFoundError", and terminate this algorithm.
12. If there are no authenticators currently available on this platform, return a DOMException whose name is "SecurityError", and terminate this algorithm.
11. Let issuedRequests be a new ordered set.
10. Let collectedClientData be a new CollectedClientData instance whose fields are:
   - challenge: The base64url encoding of publicKeyOptions.challenge
   - origin: The unicode serialization of rpId
   - hashAlg: The recognized algorithm name of the hash algorithm selected by the client for generating the hash of the serialized client data
   - tokenBinding: The Token Binding ID associated with callerOrigin, if one is available.
9. Let clientDataJSON be the JSON-serialized client data constructed from collectedClientData.
8. Let clientDataHash be the hash of the serialized client data represented by clientDataJSON.
7. If extensionId is not an authenticator extension, then continue.
6. Let clientExtensions be a new map and let authenticatorExtensions be a new map.
5. Set authenticatorExtensions[extensionId] to the base64url encoding of authenticatorExtensionInput.
4. Set clientExtensionInput to the CBOR result of running extensionId's client extension processing algorithm on clientExtensionInput. If the algorithm returned an error, continue.
3. If extensionId is not an authenticator extension, then continue.
2. Set clientExtensions[extensionId] to clientExtensionInput.
1. If extensionId is not supported by this client platform or is not an authentication extension, then continue.
0. For each extensionId -> clientExtensionInput of eachId -> clientExtensionInput of
publicKeyOptions.extensions:
   1. If extensionId is not supported by this client platform or is not an authentication extension, then continue.
   2. Set clientExtensions[extensionId] to clientExtensionInput.
   3. If extensionId is not an authentication extension, then continue.
   4. Let extensionId's client extension processing algorithm on clientExtensionInput. If the algorithm returned an error, continue.
   5. Set authenticatorExtensions[extensionId] to the base64url encoding of authenticatorExtensionInput.
   6. Let clientExtensions be a new map.
   7. If the extensions member of publicKeyOptions is present, then for each extensionId -> clientExtensionInput of
publicKeyOptions.extensions:
      1. If extensionId is not supported by this client platform or is not an authentication extension, then continue.
      2. Set clientExtensions[extensionId] to clientExtensionInput.
      3. If extensionId is not an authentication extension, then continue.
      4. Let extensionId's client extension processing algorithm on clientExtensionInput. If the algorithm returned an error, continue.
      5. Set authenticatorExtensions[extensionId] to the base64url encoding of authenticatorExtensionInput.
     6. Let clientExtensions be a new map.
   8. Let collectedClientData be a new CollectedClientData instance whose fields are:
      - challenge: The base64url encoding of options.challenge
      - origin: The serialization of callerOrigin.
      - hashAlgorithm: The recognized algorithm name of the hash algorithm selected by the client for generating the hash of the serialized client data
      - tokenBindingId: The Token Binding ID associated with callerOrigin, if one is available.
     9. Let clientDataJSON be the JSON-serialized client data constructed from collectedClientData.
    10. Let clientDataHash be the hash of the serialized client data represented by clientDataJSON.
6. If there are no authenticators currently available on this platform, return a DOMException whose name is "NotFoundError", and terminate this algorithm.
5. Let clientDataHash be the hash of the serialized client data represented by clientDataJSON.
4. Let issuedRequests be a new ordered set.
3. If the rpId member of publicKeyOptions is not present, then set rpId to callerOrigin. Otherwise:
   1. Let effectiveDomain be the callerOrigin's effective domain.
   2. If effectiveDomain is null, then return a DOMException whose name is "SecurityError" and terminate this algorithm.
   3. If rpId is not a registrable domain suffix of and is not equal to effectiveDomain, return a DOMException whose name is "NotAllowedError", and terminate this algorithm.
   4. Set rpId to the rpId.
5. Set rpId to the callerOrigin. Otherwise:
   1. If options.rpId is not a registrable domain suffix of and is not equal to effectiveDomain, return a DOMException whose name is "SecurityError", and terminate this algorithm.
   2. Set rpId to options.rpId.
   3. If options.rpId is not present, then set rpId to effectiveDomain.
   4. If options.clientExtensionInput is present, then for each extensionId -> clientExtensionInput of
      eachId -> clientExtensionInput of
publicKeyOptions.extensions:
         1. If extensionId is not supported by this client platform or is not an authentication extension, then continue.
         2. Set clientExtensions[extensionId] to clientExtensionInput.
         3. If extensionId is not an authentication extension, then continue.
         4. Let extensionId's client extension processing algorithm on clientExtensionInput. If the algorithm returned an error, continue.
         5. Set authenticatorExtensions[extensionId] to the base64url encoding of authenticatorExtensionInput.
     6. Let clientExtensions be a new map.
   7. If the extensions member of options is present, then for each extensionId -> clientExtensionInput of
publicKeyOptions.extensions:
      1. If extensionId is not supported by this client platform or is not an authentication extension, then continue.
      2. Set clientExtensions[extensionId] to clientExtensionInput.
      3. If extensionId is not an authentication extension, then continue.
      4. Let extensionId's client extension processing algorithm on clientExtensionInput. If the algorithm returned an error, continue.
      5. Set authenticatorExtensions[extensionId] to the base64url encoding of authenticatorExtensionInput.
   8. Let clientExtensions be a new map.
   9. If the extensions member of options is present, then for each extensionId -> clientExtensionInput of
publicKeyOptions.extensions:
      1. If extensionId is not supported by this client platform or is not an authentication extension, then continue.
      2. Set clientExtensions[extensionId] to clientExtensionInput.
      3. If extensionId is not an authentication extension, then continue.
      4. Let extensionId's client extension processing algorithm on clientExtensionInput. If the algorithm returned an error, continue.
      5. Set authenticatorExtensions[extensionId] to the base64url encoding of authenticatorExtensionInput.
   10. Let clientExtensions be a new map.
   11. Let clientDataJSON be the JSON-serialized client data constructed from collectedClientData.
   12. Let clientDataHash be the hash of the serialized client data represented by clientDataJSON.
   13. Let issuedRequests be a new ordered set.
   14. If the rpId member of publicKeyOptions is not present, then set rpId to callerOrigin. Otherwise:
      1. If options.rpId is not a registrable domain suffix of and is not equal to effectiveDomain, return a DOMException whose name is "SecurityError", and terminate this algorithm.
      2. Set rpId to options.rpId.
      3. If options.rpId is not present, then set rpId to effectiveDomain.
      4. If options.clientExtensionInput is present, then for each extensionId -> clientExtensionInput of
         eachId -> clientExtensionInput of
publicKeyOptions.extensions:
            1. If extensionId is not supported by this client platform or is not an authentication extension, then continue.
            2. Set clientExtensions[extensionId] to clientExtensionInput.
            3. If extensionId is not an authentication extension, then continue.
            4. Let extensionId's client extension processing algorithm on clientExtensionInput. If the algorithm returned an error, continue.
            5. Set authenticatorExtensions[extensionId] to the base64url encoding of authenticatorExtensionInput.
       6. Let clientExtensions be a new map.
   7. If the extensions member of options is present, then for each extensionId -> clientExtensionInput of
publicKeyOptions.extensions:
      1. If extensionId is not supported by this client platform or is not an authentication extension, then continue.
      2. Set clientExtensions[extensionId] to clientExtensionInput.
      3. If extensionId is not an authentication extension, then continue.
      4. Let extensionId's client extension processing algorithm on clientExtensionInput. If the algorithm returned an error, continue.
      5. Set authenticatorExtensions[extensionId] to the base64url encoding of authenticatorExtensionInput.
   8. Let clientExtensions be a new map.
   9. If the extensions member of options is present, then for each extensionId -> clientExtensionInput of
publicKeyOptions.extensions:
      1. If extensionId is not supported by this client platform or is not an authentication extension, then continue.
      2. Set clientExtensions[extensionId] to clientExtensionInput.
      3. If extensionId is not an authentication extension, then continue.
      4. Let extensionId's client extension processing algorithm on clientExtensionInput. If the algorithm returned an error, continue.
      5. Set authenticatorExtensions[extensionId] to the base64url encoding of authenticatorExtensionInput.
   10. Let clientExtensions be a new map.
   11. Let clientDataJSON be the JSON-serialized client data constructed from collectedClientData.
   12. Let clientDataHash be the hash of the serialized client data represented by clientDataJSON.
   13. Let issuedRequests be a new ordered set.
   14. If the rpId member of publicKeyOptions is not present, then set rpId to callerOrigin. Otherwise:
      1. If options.rpId is not a registrable domain suffix of and is not equal to effectiveDomain, return a DOMException whose name is "SecurityError", and terminate this algorithm.
      2. Set rpId to options.rpId.
      3. If options.rpId is not present, then set rpId to effectiveDomain.
      4. If options.clientExtensionInput is present, then for each extensionId -> clientExtensionInput of
         eachId -> clientExtensionInput of
publicKeyOptions.extensions:
            1. If extensionId is not supported by this client platform or is not an authentication extension, then continue.
            2. Set clientExtensions[extensionId] to clientExtensionInput.
            3. If extensionId is not an authentication extension, then continue.
            4. Let extensionId's client extension processing algorithm on clientExtensionInput. If the algorithm returned an error, continue.
            5. Set authenticatorExtensions[extensionId] to the base64url encoding of authenticatorExtensionInput.
perform the following steps:

1. Let credentialList be a new list.
2. For each authenticator in issuedRequests invoke the authenticators.
3. If credentialList is empty then continue.
4. In parallel, for each credential C in credentialList:
   1. If C.transports is not empty, the client SHOULD select one transport from transports. Then, using transport, invoke the authenticatorGetAssertion operation on authenticator, with rpId, clientDataHash, credentialList, and clientExtensions as parameters.
   2. Otherwise, using local configuration knowledge of the appropriate transport to use with authenticator, invoke the authenticatorGetAssertion operation on authenticator with rpId, clientDataHash, credentialList, and clientExtensions as parameters.
5. Append authenticator to issuedRequests.
14. Start a timer for adjustedTimeout milliseconds. Then execute the following steps in parallel. The task source for these tasks is the dom manipulation task source.

15. While issuedRequests is not empty, perform the following actions depending upon the adjustedTimeout timer and responses from the authenticators:
   1. Let allowCredentialDescriptorList be a new list.
   2. If options.allowCredentials is not empty, execute a platform-specific procedure to determine which, if any, public key credentials described by options.allowCredentials are bound to this authenticator, by matching with rpId, options.allowCredentials.id, and options.allowCredentials.type. Set allowCredentialDescriptorList to this filtered list.
   3. If allowCredentialDescriptorList is not empty
      1. Let distinctTransports be a new ordered set.
      2. For each credential descriptor C in allowCredentialDescriptorList, append each value, if any, of C.transports to distinctTransports.
      3. If distinctTransports is not empty
         1. The client selects one transport value from distinctTransports, possibly incorporating local configuration knowledge of the appropriate transport to use with authenticator in making its selection.
         2. Then, using transport, invoke in parallel the authenticatorGetAssertion operation on authenticator with rpId, clientDataHash, allowCredentialDescriptorList, and clientExtensions as parameters.
      4. Append authenticator to issuedRequests.
17. Start a timer for adjustedTimeout milliseconds. Then execute the following steps in parallel. The task source for these tasks is the dom manipulation task source.
   1. Let allowCredentialDescriptorList be a new list.
   2. If options.allowCredentials is not empty, execute a platform-specific procedure to determine which, if any, public key credentials described by options.allowCredentials are bound to this authenticator, by matching with rpId, options.allowCredentials.id, and options.allowCredentials.type. Set allowCredentialDescriptorList to this filtered list.
   3. If allowCredentialDescriptorList is not empty
      1. Let distinctTransports be a new ordered set.
      2. For each credential descriptor C in allowCredentialDescriptorList, append each value, if any, of C.transports to distinctTransports.
      3. If distinctTransports is not empty
         1. The client selects one transport value from distinctTransports, possibly incorporating local configuration knowledge of the appropriate transport to use with authenticator in making its selection.
         2. Then, using transport, invoke in parallel the authenticatorGetAssertion operation on authenticator with rpId, clientDataHash, allowCredentialDescriptorList, and clientExtensions as parameters.
      4. Append authenticator to issuedRequests.
1. Remove authenticator from issuedRequests.
2. For each remaining authenticator in issuedRequests invoke the authenticatorCancel operation on authenticator and remove it from issuedRequests.

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

If any authenticator indicates success, if any authenticator indicates success, if any authenticator indicates success, 1. Remove authenticator from issuedRequests.
2. Let value be a new PublicKeyCredential associated with
   global whose fields are:
   [[identifier]],
   A new ArrayBuffer, created using global's
   %ArrayBuffer%, containing the
   bytes of the returned authenticatorData
   A new ArrayBuffer, created using
   global's %ArrayBuffer%, containing the
   bytes of clientDataJSON
   signature
   A new ArrayBuffer, created using
   global's %ArrayBuffer%, containing the
   bytes of the returned signature
   clientExtensionResults
   A new AuthenticationExtensions object
   containing the extension identifier -> client
   extension output entries created by running
   each extension's client extension processing
   algorithm to create the client extension
   outputs, for each client extension in
   clientDataJSON.clientExtensions.
3. For each remaining authenticator in issuedRequests invoke
   the authenticatorCancel operation on authenticator and
   remove it from issuedRequests.
4. Return value and terminate this algorithm.

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

During the above process, the user agent SHOULD show some UI to the
user to guide them in the process of selecting and authorizing an
authenticator with which to complete the operation.
4.2. Authenticator Responses (interface AuthenticatorResponse)

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

```javascript
interface AuthenticatorResponse {
    readAttribute ArrayBuffer clientDataJSON;
    
    interface AuthenticatorResponse
    
    clientDataJSON, of type ArrayBuffer, readonly
    
    This attribute contains a JSON serialization of the client data passed to the authenticator by the client in its call to either create() or get().
    
    4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)
    
    The AuthenticatorAttestationResponse interface represents the client's request for the creation of a new public key credential. It contains information about the new credential that can be used to identify it for later use, and metadata that can be used by the Relying Party to assess the characteristics of the credential during registration.
    
        [SameObject] readAttribute ArrayBuffer attestationObject;
        
        clientDataJSON
        
        This attribute, inherited from AuthenticatorResponse, contains the JSON-serialized client data (see 5.3 Credential)
    }
    
    This attribute, inherited from AuthenticatorResponse, contains the JSON-serialized client data (see 5.3 Credential) passed to
```
4.3. Parameters for Credential Generation (dictionary PublicKeyCredentialParameters)

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

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

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

The algorithm member specifies the cryptographic signature algorithm with which the newly generated credential will be used, and thus also the type of asymmetric key pair to be generated, e.g., RSA or Elliptic Curve.
4.4. User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity)

dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
  DOMString displayName;
};

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

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

4.5. Options for Credential Creation (dictionary MakeCredentialOptions)

dictionary MakeCredentialOptions {
  required PublicKeyCredentialEntity rp;
  required PublicKeyCredentialUserEntity user;
  required BufferSource challenge;
  required sequence<PublicKeyCredentialParameters> parameters;
  unsigned long timeout;
  sequence<PublicKeyCredentialDescriptor> excludeList;
  AuthenticatorSelectionCriteria authenticatorSelection;
  AuthenticationExtensions extensions;
};

4.4. Options for Credential Creation (dictionary MakePublicKeyCredentialOptions)

dictionary MakePublicKeyCredentialOptions {
  required PublicKeyCredentialEntity rp;
  required PublicKeyCredentialUserEntity user;
  required BufferSource challenge;
  required sequence<PublicKeyCredentialParameters> pubKeyCredParams;
  unsigned long timeout;
  sequence<PublicKeyCredentialDescriptor> excludeCredentials = [];
  AuthenticatorSelectionCriteria authenticatorSelection;
  AuthenticationExtensions extensions;
};

rp, of type PublicKeyCredentialEntity

This member contains data about the relying party responsible for the request.

Its value’s name member is required, and contains the friendly name of the relying party (e.g. "Acme Corporation", "Widgets, Inc.", or "Awesome Site").

Its value’s id member specifies the relying party identifier with which the credential should be associated. If this identifier is not explicit, it will default to the ASCII serialization of the CredentialsContainer object’s relevant settings object’s origin.

user, of type PublicKeyCredentialUserEntity

This member contains data about the user account for which the relying party is requesting attestation.

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

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

Its value’s id member is required, and contains an identifier for the account, specified by the relying party. This is not meant to be displayed to the user, but is used by the relying party to control the number of credentials - an authenticator will never contain more than one credential for a given relying party under the same id.

challenge, of type BufferSource

This member contains a challenge intended to be used for generating the newly created credential’s attestation object.

parameters, of type sequence<PublicKeyCredentialParameters>

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

timeout, of type unsigned long.

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

exclusiveList, of type sequence<PublicKeyCredentialDescriptor>

This member is intended for use by Relying Parties that wish to limit the creation of multiple credentials for the same account on a single authenticator. The platform is requested to return an error if the new credential would be created on an authenticator that also contains one of the credentials enumerated in this parameter.

AuthenticatorSelection, of type AuthenticatorSelectionCriteria

This member is intended for use by Relying Parties that wish to select the appropriate authenticators to participate in the create() or get() operation.

extensions, of type AuthenticationExtensions

This member contains additional parameters requesting additional processing by the client and authenticator. For example, the caller may request that only authenticators with certain capabilities be used to create the credential, or that particular information be returned in the attestation object. Some extensions are defined in 8 WebAuthn Extensions; consult the IANA "WebAuthn Extension Identifier" registry established by [WebAuthn-Registries] for an up-to-date list of registered WebAuthn Extensions.


4.5.1. Entity Description

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

dictionary PublicKeyCredentialEntity {

DOMString id;

DOMString name;

USVString icon;

};

id, of type DOMString

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

name, of type DOMString

A human-friendly identifier for the entity. For example, this could be a company name for a relying party, or a user’s name.

icon, of type USVString

A serialized URL which resolves to an image associated with the entity. For example, this could be a user's avatar or a relying party's logo.

4.5.2. Authenticator Selection Criteria

to create the most preferred credential that it can.

timeout, of type unsigned long.

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

excludeCredentials, of type sequence<PublicKeyCredentialDescriptor>,
defaulting to None

This member is intended for use by Relying Parties that wish to limit the creation of multiple credentials for the same account on a single authenticator. The platform is requested to return an error if the new credential would be created on an authenticator that also contains one of the credentials enumerated in this parameter.

AuthenticatorSelection, of type AuthenticatorSelectionCriteria

This member is intended for use by Relying Parties that wish to select the appropriate authenticators to participate in the create() or get() operation.

extensions, of type AuthenticationExtensions

This member contains additional parameters requesting additional processing by the client and authenticator. For example, the caller may request that only authenticators with certain capabilities be used to create the credential, or that particular information be returned in the attestation object. Some extensions are defined in 8 WebAuthn Extensions; consult the IANA "WebAuthn Extension Identifier" registry established by [WebAuthn-Registries] for an up-to-date list of registered WebAuthn Extensions.

4.4.1. Public Key Entity Description (dictionary PublicKeyCredentialEntity)

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

dictionary PublicKeyCredentialEntity {

DOMString id;

DOMString name;

USVString icon;

};

id, of type DOMString

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

name, of type DOMString

A human-friendly identifier for the entity. For example, this could be a company name for a Relying Party, or a user's name.

icon, of type USVString

A serialized URL which resolves to an image associated with the entity. For example, this could be a user's avatar or a Relying Party's logo.

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

The PublicKeyCredentialUserEntity dictionary is used to supply additional user account attributes when creating a new credential.

dictionary PublicKeyCredentialUserEntity {

DOMString displayName;

};

displayName, of type DOMString

A friendly name for the user account (e.g., "John P. Smith").

4.4.3. Authenticator Selection Criteria (dictionary


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

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

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

4.6. Options for Assertion Generation (dictionary PublicKeyCredentialRequestOptions)

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

- **required BufferSource challenge;**
- **unsigned long timeout;**
- **USVString rpId;**
- **sequence<PublicKeyCredentialDescriptor> allowList = [];**
- **AuthenticationExtensions extensions;**

This optional member contains a list of credentials acceptable to the caller, in descending order of the claimed by the caller. If omitted, its value will be the CredentialsContainer object's relevant settings object's origin's effective domain.

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

4.7. Authentication Extensions (typedef AuthenticationExtensions)

typedef record<DOMString, any> AuthenticationExtensions;

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

4.8. Supporting Data Structures

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

- 4.8.1. Client data used in WebAuthn signatures (dictionary CollectedClientData)
- 4.8.2. Challenge used in WebAuthn signatures (dictionary CollectedChallenges)
- 4.8.3. Authenticator data used in WebAuthn signatures (dictionary CollectedAuthenticatorData)
- 4.8.4. T Window used in WebAuthn signatures (dictionary CollectedTWindow)
- 4.8.5. T Window used in WebAuthn signatures (dictionary CollectedTWindow)
- 4.8.6. Challenge used in WebAuthn signatures (dictionary CollectedChallenges)
- 4.8.7. Authenticator data used in WebAuthn signatures (dictionary CollectedAuthenticatorData)
- 4.8.8. T Window used in WebAuthn signatures (dictionary CollectedTWindow)
- 4.8.9. Challenge used in WebAuthn signatures (dictionary CollectedChallenges)

The client data represents the contextual bindings of both the Relying Party and the client platform. It is a key-value mapping with string-valued keys. Values may be any type that has a valid encoding in
get() methods. It mirrors the fields of the PublicKeyCredential object when referring to a credential as an input parameter to the create() or this dictionary contains the attributes that are specified by a caller as provided to the authenticator by the client, in the syntax defined by [RFC6454].

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

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

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

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

- JSON-serialized client data
- This is the UTF-8 encoding of the result of calling the initial value of JSON.stringify on a CollectedClientData dictionary.
- Hash of the serialized client data
- This is the hash (computed using hashAlg) of the JSON-serialized client data, as constructed by the client.

4.8.2. Credential Type enumeration (enum PublicKeyCredentialType)
enum PublicKeyCredentialType {
  "public-key",
};

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

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

4.8.3. Credential Descriptor (dictionary PublicKeyCredentialDescriptor)
dictionary PublicKeyCredentialDescriptor {
  required PublicKeyCredentialType type;
  required BufferSource id;
  sequence<Transports> transports;
};

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

//Users/jehodges/Documents/work/standards/W3C/webauthn/index-master-tr-d9a3e24-WD-05.txt, Top line: 1462

1462 JSON. Its structure is defined by the following Web IDL:
1463 dictionary CollectedClientData {
1464  required DOMString challenge;
1465  required DOMString origin;
1466  required DOMString hashAlg;
1467  DOMString tokenBinding;
1468  AuthenticationExtensions clientExtensions;
1469  AuthenticationExtensions authenticatorExtensions;
1470 }; The challenge member contains the base64url encoding of the challenge provided by the RP.
1471 The origin member contains the fully qualified origin of the requester, as provided to the authenticator by the client, in the syntax defined by [RFC6454].
1472 The hashAlg member is a recognized algorithm name that supports the "digest" operation, which specifies the algorithm used to compute the hash of the serialized client data. This algorithm is chosen by the client at its sole discretion.
1473 The tokenBinding member contains the base64url encoding of the Token Binding ID that this client uses for the Token Binding protocol when communicating with the Relying Party. This can be omitted if no Token Binding has been negotiated between the client and the Relying Party.
1474 The optional clientExtensions and authenticatorExtensions members contain additional parameters generated by processing the extensions passed in by the Relying Party. WebAuthn extensions are detailed in section 8 WebAuthn Extensions.
1475 This structure is used by the client to compute the following quantities:
1476 - JSON-serialized client data
1477 - This is the UTF-8 encoding of the result of calling the initial value of JSON.stringify on a CollectedClientData dictionary.
1478 - Hash of the serialized client data
1479 - This is the hash (computed using hashAlg) of the JSON-serialized client data, as constructed by the client.

4.8.2. Credential Type enumeration (enum PublicKeyCredentialType)
enum PublicKeyCredentialType {
  "public-key",
};

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

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

4.8.3. Credential Descriptor (dictionary PublicKeyCredentialDescriptor)
dictionary PublicKeyCredentialDescriptor {
  required PublicKeyCredentialType type;
  required BufferSource id;
  sequence<Transports> transports;
};

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

//Users/jehodges/Documents/work/standards/W3C/webauthn/index-master-tr-598ac41-WD-06.txt, Top line: 1631
1631 JSON. Its structure is defined by the following Web IDL:
1632 dictionary CollectedClientData {
1633  required DOMString challenge;
1634  required DOMString origin;
1635  required DOMString hashAlg;
1636  DOMString tokenBinding;
1637  AuthenticationExtensions clientExtensions;
1638  AuthenticationExtensions authenticatorExtensions;
1639 }; The challenge member contains the base64url encoding of the challenge provided by the RP.
1640 The origin member contains the fully qualified origin of the requester, as provided to the authenticator by the client, in the syntax defined by [RFC6454].
1641 The hashAlg member is a recognized algorithm name that supports the "digest" operation, which specifies the algorithm used to compute the hash of the serialized client data. This algorithm is chosen by the client at its sole discretion.
1642 The tokenBindingId member contains the base64url encoding of the Token Binding ID that this client uses for the Token Binding protocol when communicating with the Relying Party. This can be omitted if no Token Binding has been negotiated between the client and the Relying Party.
1643 The optional clientExtensions and authenticatorExtensions members contain additional parameters generated by processing the extensions passed in by the Relying Party. WebAuthn extensions are detailed in section 8 WebAuthn Extensions.
1644 This structure is used by the client to compute the following quantities:
1645 - JSON-serialized client data
1646 - This is the UTF-8 encoding of the result of calling the initial value of JSON.stringify on a CollectedClientData dictionary.
1647 - Hash of the serialized client data
1648 - This is the hash (computed using hashAlg) of the JSON-serialized client data, as constructed by the client.

4.7.2. Credential Type enumeration (enum PublicKeyCredentialType)
enum PublicKeyCredentialType {
  "public-key",
};

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

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

4.7.3. Credential Descriptor (dictionary PublicKeyCredentialDescriptor)
dictionary PublicKeyCredentialDescriptor {
  required PublicKeyCredentialType type;
  required BufferSource id;
  sequence<Transport> transports;
};

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

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

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

4.8.5. Cryptographic Algorithm Identifier (type AlgorithmIdentifier)

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

5. WebAuthn Authenticator model

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

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

For authenticators, this model defines the logical operations that they must support, and the data formats that they expose to the client and the Relying Party. However, it does not define the details of how authenticators communicate with the client, nor the data formats required for interoperability with Relying Parties. For instance, this abstract model does not define specific error codes or methods of returning them; however, it does define error behavior in terms of the needs of the client. Therefore, specific error codes are mentioned as a means of showing which error conditions must be distinguishable (or not) from each other in order to enable a compliant and secure client implementation.

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

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

Each authenticator has an AAGUID, which is a 128-bit identifier that identifies the type (e.g., make and model) of the authenticator. The AAGUID MUST be chosen by the manufacturer to be identical across all substantially identical authenticators that manufac

The primary function of the authenticator is to provide WebAuthn signatures, which are bound to various contextual data. These contextual data are observed, and added at different levels of the stack as a signature request passes from the server to the authenticator. In verifying a signature, the server checks these bindings against expected values. These contextual bindings are divided in two: Those added by the RP or the client, referred to as client data; and those added by the authenticator, referred to as the authenticator data. The authenticator also asserts additional information, termed contextual bindings, which are bound to various contextual data.

The goals of this design can be summarized as follows:

* The scheme for generating signatures should accommodate cases where the link between the client platform and authenticator is very limited, in bandwidth and/or latency. Examples include Bluetooth Low Energy and Near-Field Communication.
* The design aims to reuse as much as possible of existing encoding formats in order to aid adoption and implementation.
* The design aims to provide cryptographic signatures for two distinct purposes:
  1. An attestation signature is produced when a new credential is created, and provides cryptographic proof of certain properties of the credential and the authenticator. For instance, an attestation signature asserts the type of the authenticator (as denoted by its AAGUID) and the public key of the credential.
  2. An assertion signature is produced when the authenticatorGetAssertion method is invoked. It represents an assertion by the authenticator that the user has consented to a specific transaction, such as logging in, or completing a purchase.

The authenticator provides cryptographic signatures for two distinct purposes:

1. An attestation signature is produced when a new public key credential is created via an authenticatorMakeCredential operation. An attestation signature provides cryptographic proof of certain properties of the the authenticator and the credential. For instance, an attestation signature asserts the type (as denoted by its AAGUID) and the public key of the credential.

2. An assertion signature is produced when the authenticatorGetAssertion method is invoked. It represents an assertion by the authenticator that the user has consented to a specific transaction, such as logging in, or completing a purchase.

Thus, an assertion signature asserts that the authenticator which possesses a particular credential has established, to the best of its ability, that the human who is requesting this transaction is the same human who consented to creating that particular credential. It also provides additional information that might be useful to the caller, such as the means by which user consent was provided, and the prompt that was shown to the user by the authenticator.
The formats of these signatures, as well as the procedures for generating them, are specified below.

5.1. Authenticator data

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

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

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

<table>
<thead>
<tr>
<th>Length (in bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 SHA-256 hash of the RP ID associated with the credential.</td>
<td></td>
</tr>
<tr>
<td>1 Flags (bit 0 is the least significant bit):</td>
<td></td>
</tr>
<tr>
<td>* Bit 0: Test of User Presence (TUP) result.</td>
<td></td>
</tr>
<tr>
<td>* Bits 1-5: Reserved for future use (RFU).</td>
<td></td>
</tr>
<tr>
<td>* Bit 6: Attestation data included (AT). Indicates whether the authenticator added attestation data.</td>
<td></td>
</tr>
<tr>
<td>* Bit 7: Extension data included (ED). Indicates if the authenticator data has extensions.</td>
<td></td>
</tr>
<tr>
<td>4 Signature counter (signCount), 32-bit unsigned big-endian integer.</td>
<td></td>
</tr>
<tr>
<td>variable (if present) attestation data (if present). See 5.3.1.</td>
<td></td>
</tr>
<tr>
<td>Authentication data for data. Its length depends on the length of the credential public key and credential ID being attested.</td>
<td></td>
</tr>
<tr>
<td>variable (if present) Extension data. This is a CBOR [RFC7049] map with extension identifiers as keys, and authenticator extension outputs as values. See 8 Webcam Extensions for details.</td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

The formats of these signatures, as well as the procedures for generating them, are specified below.
If the authenticator does not include any extension data, it MUST set the ED flag in the first byte to zero, and to one if extension data is included.

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


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

5.2. Authenticator operations

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

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

5.2.1. The authenticatorMakeCredential operation

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

1. The caller’s RP ID, as determined by the user agent and the client.
2. The hash of the serialized client data, provided by the client.
3. An optional list of PublicKeyCredentialDescriptor objects provided by the Relying Party with the intention that, if any of these are returned to the authenticator, it should not create a new credential.
4. A list of PublicKeyCredential objects provided by the Relying Party, with the cryptographic algorithms normalized as per the procedure in Web Cryptography API algorithm-normalization-normalize-an-algorithm.
5. A sequence of pairs of PublicKeyCredentialType and COSEAlgorithmIdentifier requested by the Relying Party. This is created to make the most preferred credential.
6. An optional list of PublicKeyCredentialDescriptor objects provided by the Relying Party with the intention that, if any of these are shown to the authenticator, it should not create a new credential.
7. The Relying Party’s PublicKeyCredentialEntity.
8. The relying party’s PublicKeyCredentialEntity.
9. The user account’s PublicKeyCredentialUserEntity.
10. The hash of the serialized client data, provided by the client.

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

1. Check if all the supplied parameters are syntactically well-formed and of the correct length. If not, return an error code equivalent to UnknownError and terminate the operation.
2. Check if at least one of the specified combinations of PublicKeyCredentialType and cryptographic parameters is supported.
3. If not, return an error code equivalent to NotSupportedError and terminate the operation.
4. Check if a credential matching any of the supplied PublicKeyCredentialIdentifiers is present on this authenticator.
5. If so, return an error code equivalent to NotAllowedError and terminate the operation.
6. If the requireResidentKey flag is set to true and the authenticator cannot store a Client-side-resident Credential Private Key, return an error code equivalent to ConstraintError and terminate the operation.
7. Prompt the user for consent to create a new credential. The prompt for obtaining this consent is shown by the authenticator if it has

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

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

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

* Process all the supported extensions requested by the client, and generate the assertion data with attestation data as specified in 5.1 Authenticator data. Use this assertion data and the hash of the serialized client data to create an attestation object for the new credential using the procedure specified in 5.3 Credential Attestation. Generating an Attestation Object. For more details on attestation, see 5.3 Attestation.

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

5.2.2. The authenticatorGetAssertion operation

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

* The caller's RP ID, as determined by the user agent or the client.
* A list of credentials acceptable to the Relying Party (possibly filtered by the client).
* Extension data created by the client based on the extensions requested by the Relying Party, if any.

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

* Check if all the supplied parameters are syntactically well-formed and of the correct length. If not, return an error code equivalent to "NotAllowedError" and terminate the operation.
* If a list of credentials was supplied by the client, filter it by removing those credentials that are not present on this authenticator. If no list was supplied, create a list with all credentials stored for the caller's RP ID (as determined by an exact match of the RP ID).
* If no previous result existed in an empty list, return an error code equivalent to "NotAllowedError" and terminate the operation.
* Prompt the user to select a credential from among the above list. A simple, undelimited concatenation is safe to use here because the authenticator data describes its own length. The hash of the serialized client data (which potentially has a variable length) is always the last element.
* If any error occurred while generating the assertion signature, return an error code equivalent to "NotAllowedError" and terminate the operation.
that created it. It contains an attestation signature created using the key of the attesting authority (except for the case of self attestation, when it is created using the private key associated with the credential). In order to correctly interpret an attestation statement, a Relying Party needs to understand two aspects of 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 Defined Attestation Statement Formats.

Several additional attestation statement formats are defined in 7. Defined Attestation Statement Formats.

This figure illustrates only the packed attestation statement format.

This operation takes no input parameters and returns no result.

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

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

5.3. Attestation

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

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

1. The attestation statement format is the manner in which the signature is represented and the various contextual bindings are incorporated into the attestation statement by the authenticator.

In other words, this defines the syntax of the statement. Various...
existing devices and platforms (such as TPMs and the Android OS)
have previously defined attestation statement formats. This
specification supports a variety of such formats in an extensible
way, as defined in 5.3.2 Attestation Statement Formats.
2. The attestation type defines the semantics of the attestation
statement and its underlying trust model. 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 Party will also need to understand the characteristics of the authenticators that they trust, based on information they have about these authenticators. For example, the FIDO Metadata Service [FIDOMetadataService] provides one way to access such information.

5.3.1. Attestation data

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

Length (in bytes) Description
16 The AAGUID of the authenticator.
2 Byte length L of Credential ID
16 L Credential ID

variable Credential public key encoded in CBOR format. This is a CBOR map defined by the following CDDL rules:
pubKey = $pubKeyFmt

A Credential public key formatted as COSE_Key, as
//pubKey = {alg: alg, x: biguint, y: biguint}
//pubKeyTemplate = { alg: text } in CBOR format, as defined in 7.2 Packed Attestation Statement Format
pubKeyTemplate .within $pubKeyFmt

variable The credential public key encoded in COSE_Key format, as
defined in 7.2 Packed Attestation Statement Format
pubKeyTemplate = { alg: text }

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

As described above, an attestation statement format is a data format which represents the cryptographic signature of an authenticator over a set of contextual bindings. Each attestation statement format is defined by the following attributes:
- Its attestation statement format identifier.
- The set of attestation types supported by the format.
- The syntax of an attestation statement produced in this format, defined using CDDL for the extension point "attStmtFormat" defined in 5.3.4 Generating an Attestation Object.
- The procedure for computing an attestation statement in this format given the credential to be attested, the authenticator data structure containing the authenticator data for the attestation, and the hash of the serialized client data.
- The procedure for verifying an attestation statement, which takes as inputs the authenticator data structure containing the authenticator data claimed to have been used for the attestation and the hash of the serialized client data, and returns either:
  - An error indicating that the attestation is invalid, or
  - The attestation type, and the trust path of the attestation.

This trust path is either empty (in case of self-attestation), an identifier of an authenticator-specific public key (in the case of ECDAA), or a set of X.509 certificates.

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

5.3.2. Attestation Statement Formats

5.3.3. Attestation Types

WebAuthn supports multiple attestation types:

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

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

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

Attestation keys can be requested for each public key credential individually.

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

Elliptic Curve based Direct Anonymous Attestation (ECDAA)
In this case, the Authenticator receives direct anonymous attestation (DAA) credentials from a single DAA-Issuer. These DAA credentials are used along with blinding to sign the attestation object.

As described above, an attestation statement format is a data format which represents the cryptographic signature by an authenticator over a set of contextual bindings. Each attestation statement format MUST be defined using the following template:
- Attestation statement format identifier:
- Supported attestation types:
- Syntax: The syntax of an attestation statement produced in this format, defined using CDDL for the extension point "attStmtFormat" defined in 5.3.4 Generating an Attestation Object.
- Signing procedure: The signing procedure for computing an attestation statement in this format given the public key credential to be attested, the authenticator data structure containing the authenticator data for the attestation, and the hash of the serialized client data.
- Verification procedures: The procedure for verifying an attestation statement, which takes as inputs the authenticator data structure containing the authenticator data claimed to have been used for the attestation and the hash of the serialized client data, and returns either:
  - An error indicating that the attestation is invalid, or
  - The attestation type, and the trust path of the attestation.

This trust path is either empty (in case of self-attestation), an identifier of a public key (in the case of ECDAA), or a set of X.509 certificates.

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

5.3.3. Attestation Types

WebAuthn supports multiple attestation types:

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

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

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

Attestation keys can be requested for each public key credential individually.

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

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

5.3.4. Generating an Attestation Object

This section specifies the algorithm for generating an attestation object for any attestation statement format. For example, a
WebAuthn Authenticator may be capable of dynamically generating a public key credential using a particular attestation statement format, the authenticator MUST first generate the authenticator data.

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

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

\[
\text{attObj} = \begin{\{ \text{fmt: text, } \\
\text{attStmtTemplate: bytes, } \\
\text{attStmt: bytes} \}\}
\]

: Every attestation statement format must have the above fields

\text{attStmtTemplate } within \text{attStmtType}

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

\text{fmt}

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

\text{authData}

The authenticator data used to generate the attestation statement.

\text{attStmt}

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

5.3.5. Security Considerations

5.3.5.1. Privacy

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

* A WebAuthn Authenticator manufacturer may choose to ship all of their devices with the same (or a fixed number of) attestation key(s) (called Basic Attestation). This will anonymize the user at the risk of not being able to revoke a particular attestation key should its WebAuthn Authenticator be compromised.

* A WebAuthn Authenticator may be capable of dynamically generating different attestation keys (and requesting related certificates) per origin (following the Privacy CA approach). For example, a WebAuthn Authenticator can ship with a master attestation key (and certificate), and combined with a cloud operated privacy CA, can dynamically generate per origin attestation keys and attestation certificates.

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

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

\[
\text{attObj} = \begin{\{ \text{fmt: text, } \\
\text{attStmtTemplate: bytes, } \\
\text{attStmt: bytes} \}\}
\]

: Every attestation statement format must have the above fields

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

\text{fmt}

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

\text{authData}

The authenticator data used to generate the attestation statement.

\text{attStmt}

The attestation statement constructed above. The syntax of this is defined by the attestation statement format used.
A WebAuthn Authenticator can implement Elliptic Curve based direct anonymous attestation (see [FIDOEcdaaAlgorithm]). Using this scheme, the authenticator generates a blinded attestation signature. This allows the Relying Party to verify the attestation signature using the ECDAA-Issuer public key, but the attestation signature does not serve as a global correlation handle.

5.3.5.2. Attestation Certificate and Attestation Certificate CA Compromise

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

A WebAuthn Authenticator attestation certificate must be revoked by the issuing CA if its key has been compromised. A WebAuthn Authenticator manufacturer may need to ship a firmware update and inject new attestation keys and certificates into already manufactured WebAuthn Authenticators, if the exposure was due to a firmware flaw. The process by which this happens is out of scope for this specification.

If the WebAuthn Authenticator manufacturer does not have the capability, then it may not be possible for Relying Parties to trust any further attestation statements from the affected WebAuthn Authenticators.

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

If an ECDAA attestation key has been compromised, it can be added to the RogueList (i.e., the list of revoked authenticators) maintained by the related ECDAA-Issuer. The Relying Party should verify that an authenticator belongs to the RogueList when performing ECDAA-Verify (see section 3.5 in [FIDOEcdaaAlgorithm]). For example, the FIDO Metadata Service ([FIDOMetadataService]) provides one way to access such information.

5.3.5.3. Attestation Certificate Hierarchy

A 3-tier hierarchy for attestation certificates is recommended (i.e., Attestation Root, Attestation Issuing CA, Attestation Certificate). It is also recommended that for each WebAuthn Authenticator device line (i.e., model), a separate issuing CA is used to help facilitate isolating problems with a specific version of a device. If the attestation root certificate is not dedicated to a single WebAuthn Authenticator device line (i.e., AAGUID), the AAGUID should be specified in the attestation certificate itself, so that it can be verified against the authenticator data.

6. Relying Party Operations

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

6. Relying Party Operations

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

6.1. Registering a new credential

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

1. Perform JSON deserialization on the clientDataJSON field of the AuthenticatorAttestationResponse object to extract the client data.
   - ClientDataJSON is defined as collected during the credential creation ceremony.

2. Verify that the challenge in C matches the challenge that was sent to the authenticator in the create() call.

3. Verify that the origin in C matches the Relying Party's origin.

4. Verify that the tokenBinding in C matches the Token Binding ID for the TLS connection over which the attestation was obtained.

5. Verify that the clientExtensions in C is a proper subset of the extensions requested by the RP and that the authenticatorExtensions in C is also a proper subset of the extensions requested by the RP.

6. Compute the hash of clientDataJSON using the algorithm identified by C.hashAlg.

7. Perform CBOR decoding on the attestationObject field of the AuthenticatorAttestationResponse structure to obtain the attestation statement format fmt, the authenticator data authData, and the attestation statement attStmt.

8. Verify that the RP ID hash in authData is 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 [FIDO_MetadataService] provides one way to obtain such information, using the AAGUID in the attestation data provided 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 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 credential as one with self-attestation (see 5.3.3 Attestation Types). If doing so, the Relying Party is asserting that there is cryptographic proof that the public key credential has been generated by a particular authenticator model. See [FIDOSecRef] and [UAFProtocol] for a more detailed discussion.
15. If verification of the attestation statement failed, the Relying Party MUST fail the registration ceremony.

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

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

6.2. Verifying an authentication assertion

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

1. Using credential’s id attribute (or the corresponding rawId, if base64url encoding is inappropriate for your use case), look up the corresponding public credential.

2. Let cData, aData and sig denote the value of credential’s response’s clientDataJSON, authenticatorData, and signature respectively.

3. Perform JSON deserialization on cData to extract the client data C used for the signature.

4. Verify that the challenge member of C matches the challenge that was sent to the authenticator in the PublicKeyCredentialRequestOptions passed to the get() call.

5. Verify that the origin member of C matches the Relying Party’s origin.

6. Verify that the tokenBinding member of C (if present) matches the Token Binding ID for the TLS connection over which the signature was obtained.

7. Verify that the clientExtensions member of C is a proper subset of the extensions requested by the Relying Party and that the authenticatorExtensions in C is also a proper subset of the extensions requested by the Relying Party.

8. Verify that the RP ID hash in aData is the SHA-256 hash of the RP ID expected by the Relying Party.

9. Let hash be the result of computing a hash over the CData using the algorithm represented by the hashAlg member of C.

10. Using the credential public key looked up in step 1, verify that sig is a valid signature over the binary concatenation of aData and hash.

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

7. Defined Attestation Statement Formats

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

7.1. Attestation Statement Format Identifiers

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

Attestation statement format identifiers SHOULD be registered per [WebAuthn-Registries] “Registries for Web Authentication (WebAuthn)”. All registered attestation statement format identifiers are unique amongst themselves as a matter of course.
Unregistered attestation statement format identifiers SHOULD use lowercase reverse domain-name naming, using a domain name registered by the developer, in order to assure uniqueness of the identifier. All attestation statement format identifiers MUST be a maximum of 32 octets in length and MUST consist only of printable USASCII characters, excluding backslash and doublequote, i.e., VCHAR as defined in [RFC53234] but without \%x22 and \%x5C. Note: This means attestation statement format identifiers based on domain names MUST incorporate only LDH Labels [RFC5280]. Implementations MUST match WebAuthn attestation statement format identifiers in a case-sensitive fashion.

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

7.2. Packed Attestation Statement Format

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

Attestation statement format identifier

packed

Attestation types supported

All

Syntax

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

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

packedStmtFormat =
  algs: rsaAlgName / eccAlgName,
  sig: bytes,
  x5c: [ attestCert: bytes, *(caCert: bytes) ]
```

The semantics of the fields are as follows:

alg

A text string containing the name of the algorithm used to generate the attestation signature. The types rsaAlgName and eccAlgName are as defined in 5.3.1 Attestation data. "ED256" and "ED512" refer to algorithms defined in [FIDOEd256Algorithm].

sig

A byte string containing the attestation signature.

x5c

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

ecdaaKeyId

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

Signing procedure

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

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

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

If ECDAA is in use, the authenticator produces sig by concatenating authenticatorData and clientDataHash, and signing the result using an ECDAA-Issuer public key (see above).

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

Verification procedure

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

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

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

+ Verify that sig is a valid signature over the concatenation of authenticatorData and clientDataHash using the attestation public key in x5c with the algorithm specified in alg.
+ Verify that x5c meets the requirements in 7.2.1 Packed attestation statement certificate requirements.
+ If x5c contains an extension with OID 1 3 6 1 4 1 45724 1 14 (id-fido-gen-ce-aaguid) verify that the value of this extension matches the AAGUID in authenticatorData.
+ If successful, return attestation type Basic and trust path x5c.

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

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

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

ecdaaKeyId

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

Signing procedure

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

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

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

If ECDAA is in use, the authenticator produces sig by concatenating authenticatorData and clientDataHash, and signing the result using an ECDAA-Issuer public key (see above).

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

Verification procedure

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

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

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

+ Verify that sig is a valid signature over the concatenation of authenticatorData and clientDataHash using the attestation public key in x5c with the algorithm specified in alg.
+ Verify that x5c meets the requirements in 7.2.1 Packed attestation statement certificate requirements.
+ If x5c contains an extension with OID 1 3 6 1 4 1 45724 1 14 (id-fido-gen-ce-aaguid) verify that the value of this extension matches the AAGUID in authenticatorData.
+ If successful, return attestation type Basic and trust path x5c.

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

+ Verify that sig is a valid signature over the concatenation of authenticatorData and clientDataHash using ECDAA-Verify with ECDAA-Issuer public key identified by ecdaaKeyId (see [FIDOEcdaaAlgorithm]).
7.2.1. Packed attestation statement certificate requirements

The attestation certificate MUST have the following fields/extensions:

- Version must be set to 3.
- Subject field MUST be set to:
  - Subject-C: Country where the Authenticator vendor is incorporated
  - Subject-O: Legal name of the Authenticator vendor
  - Subject-OU: Authenticator Attestation
  - Subject-CN: No stipulation.
- If the related attestation root certificate is used for multiple authenticator models, the Extension OID 1 3 6 1 4 1 45724 1 1 4 (id-fido-gen-ce-aaguid) MUST be present, containing the AAGUID as Subject-CN.
- The Basic Constraints extension MUST have the CA component set to false.
- An Authority Information Access (AIA) extension with entry id-ad-ocsp and a CRL Distribution Point extension [RFC5280] are both optional as the status of many attestation certificates is available through authenticator metadata services. See, for example, the FIDO Metadata Service [FIDOMetadataService].

7.3. TPM Attestation Statement Format

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

Attestation statement format identifier

tpm

Attestation types supported

Privacy CA, ECDAA

Syntax

The syntax of a TPM Attestation statement is as follows:

```plaintext
attsStmtType // = ( 
  fmt: "tpm",
  attStmt: tpmStmtFormat
) 
```

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

The syntax of a TPM Attestation statement is as follows:

```plaintext
attsStmtType // = ( 
  fmt: "tpm",
  attStmt: tpmStmtFormat
) 
```

```plaintext
tpmStmtFormat = ( 
  ver: "2.0",
  { alg: rsaAlgName / eccAlgName, 
    x5c: [ aikCert: bytes, * (caCert: bytes) ] 
  } 
) 
```
The semantics of the above fields are as follows:

- **ver**: The version of the TPM specification to which the signature conforms.
- **alg**: The name of the algorithm used to generate the attestation signature. The types rsaAlgName and eccAlgName are as defined in 5.3.1 Attestation data. The types "ED256" and "ED512" refer to the algorithms specified in [FIDOEcdaaAlgorithm].
- **sig**: The attestation signature, in the form of a TPMT_SIGNATURE structure as specified in [TPMv2-Part2] section 11.3.4.
- **certInfo**: The TPMS_ATTEST structure over which the above signature was computed, as specified in [TPMv2-Part2] section 10.12.8.
- **pubArea**: The TPMT_PUBLIC structure (see [TPMv2-Part2] section 12.2.4) used by the TPM to represent the credential public key.

### Signing procedure

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

- Concatenate authenticatorData and clientDataHash to form attToBeSigned.
- Generate a signature using the procedure specified in [TPMv2-Part3] Section 18.2, using the attestation private key and setting the qualifyingData parameter to attToBeSigned.
- Set the pubArea field to the public area of the credential public key, the certInfo field to the output parameter of the same name, and the sig field to the signature obtained from the above procedure.

### Verification procedure

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

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

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

Attestation types supported

Basic

Syntax

An Android key attestation statement consists simply of the

Attestation types supported

Basic

Syntax

An Android key attestation statement consists simply of the

7.5. Android SafetyNet Attestation Statement Format

When the authenticator in question is a platform-provided Authenticator

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

SafetyNet API. In this case the authenticator data is completely

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

running on the Android platform) and the attestation statement only

provides some statements about the health of the platform and the

Identity of the calling application.

Attestation statement format identifier

android-safetynet

Attestation statement format identifier

android-safetynet

Syntax

An Android key attestation statement consists simply of the

Attestation types supported

Basic

Syntax

An Android key attestation statement consists simply of the

7.5. Android SafetyNet Attestation Statement Format

When the authenticator in question is a platform-provided Authenticator

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

SafetyNet API. In this case the authenticator data is completely

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

running on the Android platform) and the attestation statement only

provides some statements about the health of the platform and the

identity of the calling application.

Attestation statement format identifier

android-safetynet

Attestation statement format identifier

android-safetynet

Syntax

An Android key attestation statement consists simply of the

Attestation types supported

Basic

Syntax

An Android key attestation statement consists simply of the

7.5. Android SafetyNet Attestation Statement Format

When the authenticator in question is a platform-provided Authenticator

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

SafetyNet API. In this case the authenticator data is completely

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

running on the Android platform) and the attestation statement only

provides some statements about the health of the platform and the

identity of the calling application.

Attestation statement format identifier

android-safetynet

Attestation statement format identifier

android-safetynet
Syntax

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

$$\text{attStmt: } \text{safetynetStmtFormat}$$

The semantics of the above fields are as follows:

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

response: Is a JWS [RFC7515] object (see SafetyNet online documentation) in Compact Serialization.

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

7.6. FIDO U2F Attestation Statement Format

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

Attestation statement format identifier fido-u2f

Attestation types supported Basic, self attestation

Syntax

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

$$\text{attStmt: } \text{fido-u2fStmtFormat}$$

The semantics of the above fields are as follows:

ver: The version number of Google Play Services responsible for providing the FIDO U2F API.

response: The value returned by the above FIDO U2F API. This value is a JSON object.

fmt: The attestation certificate format identifier.

attStmtType: The attestation certificate format identifier.

attStmt: The attestation certificate.
Web Authentication API, can be extended to suit particular use cases.

The mechanism for generating public key credentials, as well as to-be-signed data constructed above.

Claimed credential ID and the claimed credential public key.

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

Verification procedure verification is performed as follows:

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

Verification procedure verification is performed as follows:

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

8. WebAuthn Extensions

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

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

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

Client extension processing for registration extensions and authentication extensions.

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

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

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

8.1. Extension Identifiers

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

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

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

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

Extensions that may exist in multiple versions should take care to
include a version in their identifier. In effect, different versions
are thus treated as different extensions, e.g., myCompany_extension.01

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

8.2. Defining extensions

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

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

8.3. Extending request parameters

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

The IANA "WebAuthn Extension Identifier" registry established by
[WebAuthn-Registries] should be consulted for an up-to-date list of
registered WebAuthn Extensions.
A Relying Party simultaneously requests the use of an extension and sets its client extension input by including an entry in the extensions option to the create() or get() call. The entry key is the extension identifier and the value is the client extension input. 

As 5.1 Authenticator data, the CBOR authenticator extension input value of each processed authenticator extension is recorded as a dictionary in the client data with the key extenExtensions. 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.

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.

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.

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 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. In case 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.

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 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 extension input, 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.

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

8.6. Example Extension

This section is not normative.

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

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

```
var assertionPromise = navigator.credentials.get({
  challenge: "SGFufIFNvbG8gc2hvdCBmaXJzdC4",
  allowCredentials: [{ type: 'Point', extensions: { 'webauthnExample_geo': true } }
});
```

This extension specifies that the location shall be encoded as a two-element array of floating point numbers, encoded with CBOR. An authenticator does this by including it in the authenticator data. As an example, authenticator data may be as follows (notation taken from RFC 7594): [3112]

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

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

The extension also requires the client to set the authenticator to specify its geolocation in the authenticator extension output, if known. The extension e.g. specifies that the location shall be encoded as a two-element array of floating point numbers, encoded with CBOR. An authenticator does this by including it in the authenticator data. An example, authenticator data may be as follows (notation taken from RFC 7594):

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

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

9. Defined Extensions

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

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

9.1. FIDO AppId Extension (appid)

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

Extension identifier
appid

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

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

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

Authenticator extension input
None.

Authenticator extension processing
None.

Authenticator extension output
None.

9.2. Simple Transaction Authorization Extension (txAuthSimple)

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

Extension identifier
txAuthSimple

Client extension input
A single JSON string prompt.

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

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

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

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

agents targeting broad interoperability.

9.1. FIDO AppId Extension (appid)

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

Extension identifier
appid

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

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

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

Authenticator extension input
None.

Authenticator extension processing
None.

Authenticator extension output
None.

9.2. Simple Transaction Authorization Extension (txAuthSimple)

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

Extension identifier
txAuthSimple

Client extension input
A single JSON string prompt.

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

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

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

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

Each AAGUID corresponds to an authenticator model that is acceptable to the Relying Party for this credential creation.

typedef sequence<AAGUID> AuthenticatorSelectionList;

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

taxAuthGeneric

Client extension input

A CBOR map defined as follows:

taxAuthGenericArg = {
    "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

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.

 typedef BufferSource AAGUID;

typedef BufferSource AAGUID;

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

taxAuthGeneric

Client extension input

A CBOR map defined as follows:

taxAuthGenericArg = {
    "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;

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

typedef BufferSource AAGUID;
Client extension processing

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

Client extension output

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

Authenticator extension input

None.

Authenticator extension processing

None.

Authenticator extension output

None.

9.5. Supported Extensions Extension (exts)

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

Extension identifier

None.

9.5. Supported Extensions Extension (exts)

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

Authenticator extension output

None, except creating the authenticator extension input from the

Extension identifier

None, except creating the authenticator extension input from the

Client extension processing

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

Client extension output

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

Authenticator extension input

None.

Authenticator extension processing

None.

Authenticator extension output

None.

9.5. Supported Extensions Extension (exts)

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

Extension identifier

None.

9.5. Supported Extensions Extension (exts)

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

Authenticator extension output

None, except creating the authenticator extension input from the

Extension identifier

None, except creating the authenticator extension input from the

Client extension processing

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

Client extension output

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

Authenticator extension input

None.

Authenticator extension processing

None.

Authenticator extension output

None.

9.5. Supported Extensions Extension (exts)

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

Extension identifier

None.

9.5. Supported Extensions Extension (exts)

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

Authenticator extension output

None, except creating the authenticator extension input from the

Extension identifier

None, except creating the authenticator extension input from the

Client extension processing

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

Client extension output

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

Authenticator extension input

None.

Authenticator extension processing

None.

Authenticator extension output

None.

9.5. Supported Extensions Extension (exts)

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

Extension identifier

None.

9.5. Supported Extensions Extension (exts)

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

Authenticator extension output

None, except creating the authenticator extension input from the

Extension identifier

None, except creating the authenticator extension input from the

Client extension processing

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

Client extension output

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

Authenticator extension input

None.

Authenticator extension processing

None.

Authenticator extension output

None.

9.5. Supported Extensions Extension (exts)

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

Extension identifier

None.

9.5. Supported Extensions Extension (exts)

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

Authenticator extension output

None, except creating the authenticator extension input from the

Extension identifier

None, except creating the authenticator extension input from the

Client extension processing

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

Client extension output

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

Authenticator extension input

None.

Authenticator extension processing

None.

Authenticator extension output

None.

9.5. Supported Extensions Extension (exts)

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

Extension identifier

None.

9.5. Supported Extensions Extension (exts)

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

Authenticator extension output

None, except creating the authenticator extension input from the

Extension identifier

None, except creating the authenticator extension input from the

Client extension processing

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

Client extension output

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

Authenticator extension input

None.

Authenticator extension processing

None.

Authenticator extension output

None.

9.5. Supported Extensions Extension (exts)

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

Extension identifier

None.

9.5. Supported Extensions Extension (exts)

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

Authenticator extension output

None, except creating the authenticator extension input from the

Extension identifier

None, except creating the authenticator extension input from the

Client extension processing

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

Client extension output

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

Authenticator extension input

None.

Authenticator extension processing

None.

Authenticator extension output

None.

9.5. Supported Extensions Extension (exts)

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

Extension identifier

None.

9.5. Supported Extensions Extension (exts)

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

Authenticator extension output

None, except creating the authenticator extension input from the

Extension identifier

None, except creating the authenticator extension input from the

Client extension processing

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

Client extension output

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

Authenticator extension input

None.

Authenticator extension processing

None.

Authenticator extension output

None.

9.5. Supported Extensions Extension (exts)

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

Extension identifier

None.

9.5. Supported Extensions Extension (exts)

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

Authenticator extension output

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

Client extension output

Returns a JSON string containing the base64url encoding of the
authenticator extension output

Authenticator extension input

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

Authenticator extension processing

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

Authenticator extension output

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

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

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

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

Example for authenticator data containing one UVI extension

1. Example for authenticator data containing one UVI extension

```
81 00 00 01 63 76 69 58 20
```

1. Example for authenticator data containing one UVI extension

```
81 00 00 01 63 76 69 58 20
```

9.7. Location Extension (loc)

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

Extension identifier

loc

Client extension input

The Boolean value true to indicate that this extension is
The Boolean value true to indicate that this extension is a user verification method.

### 9.8. User Verification Method Extension (uvm)

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

Extension identifier

**uvm**

Client extension input

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

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

Client extension output

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

Authenticator extension input

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

Authenticator extension processing

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

Authenticator extension output

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

```
uvmFormat = [ 1*3 uvmEntry ]
uvmEntry = [ userVerificationMethod: uint .size 4,
    keyProtectionType: uint .size 2,
    matcherProtectionType: uint .size 2
  ]
```

The semantics of the fields in each uvmEntry are as follows:

- `userVerificationMethod`: The authentication method/factor used by the authenticator to verify the user. Available values are defined in [FIDOReg], "User Verification Methods" section.
- `keyProtectionType`: The method used by the authenticator to protect the FIDO registration private key material. Available values are defined in [FIDOReg], "Key Protection Types" section.
- `matcherProtectionType`: The method used by the authenticator to protect the matcher that performs user verification. Available values are defined in [FIDOReg], "Matcher Protection Types" section.

If >3 factors can be used in an authentication instance the authenticator vendor must select the 3 factors it believes will be most relevant to the Server to include in the UVM.

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

```
81  -- [RP ID] hash (32 bytes)
00 00 00 01  -- (initial) signature counter
A1  -- all public key alg etc.
63  -- extension: CBOR map of one element
76 76 6d  -- Key 1: CBOR text string of 3 bytes
82  -- Value 1: CBOR array of length 2 indicating two factor
83  -- Item 1: CBOR array of length 3
02  -- Subitem 1: CBOR integer for User Verification Method
Fingerprint
```

Example for authenticator data containing one UVM extension for a multi-factor authentication instance where 3 factors were used:

```
81  -- [RP ID] hash (32 bytes)
00 00 00 01  -- (initial) signature counter
A1  -- all public key alg etc.
63  -- extension: CBOR map of one element
A1  -- Key 1: CBOR text string of 3 bytes
82  -- "uvm" = UTF-8 encoded string
83  -- Value 1: CBOR array of length 2 indicating two factor
83  -- Item 1: CBOR array of length 3
02  -- Subitem 1: CBOR integer for User Verification Method
Fingerprint
```
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-Registrations].

* 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 attestation statement 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 a 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-Registrations].

* 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

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-Registrations].

* 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 attestation statement 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 a 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-Registrations].

* 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
Specification Document: Section 9.3 Generic Transaction

WebAuthn operation.

which user verification methods (factors) were used for the verification method extension returns to the Webauthn relying party extension enables use of a user verification method. The user extension enables use of a user verification method. The user experience around credential creation.

Specification Document: Section 9.4 Authenticator Selection

WebAuthn Extension Identifier: authnSel

Description: The location registration extension and authentication subject to user consent.

WebAuthn Extension Identifier: loc

Description: This registration extension enables the relying party to determine which extensions the authenticator supports. The extension data is a list (CBOR array) of extension identifiers, encoded as UTF-8 Strings. This extension is added automatically by the authenticator. This extension can be added to attestation statements.

Specification Document: Section 9.5 Supported Extensions Extension (exts) of this specification

WebAuthn Extension Identifier: exts

Description: This registration extension enables use of a user verification index. The user verification index is a value uniquely identifying a user verification data record. The UVI data can be used by servers to understand whether an authentication was authorized by the exact same biometric data as the initial key generation. This allows the detection and prevention of "friendly fraud".

Specification Document: Section 9.6 User Verification Index

Extension (uvi) of this specification

WebAuthn Extension Identifier: uvi

Description: This registration extension and authentication extension enables use of a user verification index. The user verification index is a value uniquely identifying a user verification data record. The UVI data can be used by servers to understand whether an authentication was authorized by the exact same biometric data as the initial key generation. This allows the detection and prevention of "friendly fraud".

Specification Document: Section 9.7 Location Extension (loc) of this specification

WebAuthn Extension Identifier: loc

Description: This registration extension enables use of a user verification method. The user verification method extension returns to the Webauthn relying party which user verification methods (factors) were used for the WebAuthn operation.

Specification Document: Section 9.8 User Verification Method

Extension (uvm) of this specification

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 Algorithmic registry (IANA-COSE-ALGS-REG).

Name: RS384

Value: -257

Description: RS384

Recommended: No

Reference: Section 8.2 of RFC8017

Value: -258

Description: RS256

Recommended: No

Reference: Section 8.2 of RFC8017

Value: -259

Description: RS512

Recommended: No

Reference: Section 8.2 of RFC8017
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 smartphone. Other authenticator types are also supported by this API, subject to implementation by the platform. For instance, this flow also works without modification for the case of an authenticator that is embedded in the client platform. The flow also works for the case of an authenticator without its own display (similar to a smart card) subject to specific implementation considerations. Specifically, the client platform needs to display any prompts that would otherwise be shown by the authenticator, and the authenticator needs to allow the client platform to enumerate all the authenticator’s credentials so that the client can have information to show appropriate prompts.

11.1. Registration

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

1. The user visits example.com, which serves up a script. At this point, the user must already be logged in using a legacy username and password, or additional authenticator, or other means acceptable to the Relying Party.

2. The Relying Party script runs the code snippet below.

```javascript
// Relying Party:
```

```javascript

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:

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

```javascript
var publicKey = {
  name: "Acme"
};
```

var publicKey = {
  challenge: Uint8Array.from(window.atob("PGifxAoBwCkWkm4b1CiIl5otCphiIh6MijdjbWFiToAr+", c=c.charCodeAt(0)),
  // Relying Party:
  rp: {
    name: "Acme"
  },
  // Server:
  "Recommended: No"
};
```

11.1. Registration

This section is not normative.

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 smartphone. 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 roamin authenticators.

1. The user visits example.com, which serves up a script. At this point, the user must already be logged in using a legacy username and password, or additional authenticator, or other means acceptable to the Relying Party. 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 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:

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

```javascript
var publicKey = {
  challenge: Uint8Array.from(window.atob("PGifxAoBwCkWkm4b1CiIl5otCphiIh6MijdjbWFiToAr+", c.charCodeAt(0)),
  // Relying Party:
  rp: {
    name: "Acme"
  },
  // Server:
  "Recommended: No"
};
```

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.
11.2. Registration Specifically with Platform Authenticator

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

1. The user visits example.com and clicks on the login button, which redirects the user to login.example.com.
2. The user enters a username and password to log in. After successful login, the user is redirected back to example.com.
3. The Relying Party script runs the code snippet below.
4. The user agent asks the user whether they are willing to register with the Relying Party using an available platform authenticator.
5. If the user is not willing, terminate this flow.
6. The user is shown appropriate UI and guided in creating a credential using one of the available platform authenticators. Upon successful credential creation, the RP script conveys the new credential to the server.
7. If (!PublicKeyCredential.isPlatformAuthenticatorAvailable) { /* Platform not capable of the API. Handle error */ }
   PublicKeyCredential.isPlatformAuthenticatorAvailable()
   .then(function (userIntent) { if (userIntent) {
   var publicKeyOptions = { /* Public key credential creation options. */
   type: "public-key",
   alg: -257 // Value registered by this specification for "RS256"
   };
   navigator.credentials.create({ publicKey:
   // Note: The following call will cause the authenticator to display UI. 
   navigator.credentials.create({ publicKeyOptions, extensions: {"webauthn.location": true} // Include location information
   },
   // in attestation
   includeCredentials: [], // No exclude list of PKCredDescriptors
   pubkeyCredentialParameters: [
   { type: "public-key",
     algorithm: "RS256"
   },
   { type: "public-key",
     algorithm: "ES256"
   }],
   // No acceptable authenticator or user refused consent. Handle appropriately
   )
   // Create and register credentials.
   return navigator.credentials.create("public-key": publicKeyOptions);
   } else {
});
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 acceptable credentials, then the sample code for performing such an authentication might look like this:

```javascript
var options = {
  challenge: new TextEncoder().encode("climb a mountain"),
  timeout: 600000, // 1 minute
  allowList: [{ type: "public-key" }]
};
```

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

```javascript
navigator.credentials.get("publicKey: options )
.then(function(credential) {
  // Send assertion to server for verification
  catch(function(err) {
    // No acceptable credential or user refused consent. Handle appropriately.
  });
});
```

### 11.3. Authentication

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

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

```javascript
var options = {
  challenge: new TextEncoder().encode("climb a mountain"),
  timeout: 600000, // 1 minute
  allowCredentails: [{ type: "public-key" }]
};
```

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

```javascript
if (!PublicKeyCredential { /* Platform not capable. Handle error. */ })
var encoder = new TextEncoder();
var acceptableCredential1 = {
  type: "public-key",
  id: encoder.encode("!!!!!hi there!!!!!!!")
};
var acceptableCredential2 = {
  type: "public-key",
  id: encoder.encode("roses are red, violets are blue")
};
var options = {
  challenge: encoder.encode("climb a mountain"),
  timeout: 60000, // 1 minute
  allowList: [acceptableCredential1, acceptableCredential2];
};
```

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

```javascript
if (!PublicKeyCredential { /* Platform not capable. Handle error. */ })
var encoder = new TextEncoder();
var acceptableCredential1 = {
  type: "public-key",
  id: encoder.encode("!!!!!hi there!!!!!!!")
};
var acceptableCredential2 = {
  type: "public-key",
  id: encoder.encode("roses are red, violets are blue")
};
```

альные возможности, при которых декоммиссия аутентификации может быть желательна. Вот несколько примеров, которые могут быть реализованы на серверной стороне и не требуют поддержки API.

* **Возможность #1** — пользователь сообщает о потере атрибута.
* **Возможность #2** — сервер отменяет атрибут на стороне пользователя.
* **Возможность #3** — пользователь удаляет атрибут с устройства.

В случае, если сказка не требует поддержки API, реализовать следующий пример можно следующим образом.

```javascript
if (!PublicKeyCredential { /* Platform not capable. Handle error. */ })
var encoder = new TextEncoder();
var acceptableCredential1 = {
  type: "public-key",
  id: encoder.encode("!!!!!hi there!!!!!!!")
};
var acceptableCredential2 = {
  type: "public-key",
  id: encoder.encode("roses are red, violets are blue")
};
```

8. **Decommissioning**

11.3. Decommissioning

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

* **Possibility #1** — user reports the credential as lost.
* **Possibility #2** — server deregisters the credential due to inactivity.
* **Possibility #3** — user deletes the credential from the device.

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

```javascript
if (!PublicKeyCredential { /* Platform not capable. Handle error. */ })
var encoder = new TextEncoder();
var acceptableCredential1 = {
  type: "public-key",
  id: encoder.encode("!!!!!hi there!!!!!!!")
};
var acceptableCredential2 = {
  type: "public-key",
  id: encoder.encode("roses are red, violets are blue")
};
```

**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.
* **Possibility #2** — server deregisters the credential due to inactivity.
* **Possibility #3** — user deletes the credential from the device.

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

```javascript
if (!PublicKeyCredential { /* Platform not capable. Handle error. */ })
var encoder = new TextEncoder();
var acceptableCredential1 = {
  type: "public-key",
  id: encoder.encode("!!!!!hi there!!!!!!!")
};
var acceptableCredential2 = {
  type: "public-key",
  id: encoder.encode("roses are red, violets are blue")
};
```
Terms defined by this specification

- AAGUID, in 9.4
- algorithm, in 4.3
- allowList, in 4.6
- Assertion, in 3
- assertion signature, in 5
- Attachment, in 4.5.3
- attachment, in 4.5.2
- attachment modality, in 4.5.3
- Attestation, in 3
- Attestation Certificate, in 3
- Attestation data, in 5.3.1
- Attestation Information, in 3
- attestation key pair, in 3
- attestationObject, in 4.2.1
- attestation objects, in 3
- attestation private key, in 3
- attestation public key, in 3
- attestation signature, in 5
- attestation statement format, in 5.3
- attestation statement format identifier, in 7.1
- attestation type, in 5.3
- Authentication, in 3
- Authentication Assertion, in 3
- authentication extension, in 8
- AuthenticationExtensions
  + definition of, in 4.7
- Authenticator, in 3
- AuthenticatorAssertionResponse, in 4.2.2
- AuthenticatorAttestationResponse, in 4.2.1
- authenticatorCancel, in 5.2.3
- authenticator data, in 5.3.1
- authenticatorData, in 4.2.2
- authenticator data claimed to have been used for the attestation, in 5.3.2
- authenticator data for the attestation, in 5.3.2
- authenticator extension, in 8
- authenticator extension input, in 6.3
- authenticator extension output, in 6.3
- Authenticator extension processing, in 8.5
- AuthenticatorExtensions, in 4.5.1
- authenticatorGetAssertion, in 5.2.2
- authenticatorMakeCredential, in 5.2.1
- AuthenticatorResponse, in 4.2
- AuthenticatorSelection, in 4.5
- AuthenticatorSelectionCriteria, in 4.5.2
- AuthenticatorSelectionList, in 9.4

- Authorization Gesture, in 3
- Base64url Encoding, in 2.1
- Basic Attestation, in 5.3.3
- Biometric Recognition, in 3
- Blob, in 4.8.4
- CBOR, in 2.1
- Ceremony, in 3
- challenge
  + dict-member for MakeCredentialOptions, in 4.5
+ dict-member for PublicKeyCredentialRequestOptions, in 4.6
+ dict-member for CollectedClientData, in 4.8.1
- Client, in 3

- AAGUID, in 9.4
- alg, in 4.3
- allowCredentials, in 4.5
- Assertion, in 3
- assertion signature, in 5
- attachment modality, in 4.4.4
- Attestation, in 3
- Attestation Certificate, in 3
- Attestation data, in 5.3.1
- assertion key pair, in 3
- assertionObject, in 4.2.1
- assertion objects, in 3
- assertion private key, in 3
- assertion public key, in 3
- assertion signature, in 5
- attestation statement format, in 5.3
- attestation statement format identifier, in 7.1
- attestation type, in 5.3
- Authentication, in 3
- Authentication Assertion, in 3
- authentication extension, in 8
- AuthenticationExtensions
  + definition of, in 4.6
  + (typedef), in 4.5
- Authenticator, in 3
- AuthenticatorAssertionResponse, in 4.2.2
- AuthenticatorAttestationResponse, in 4.2.1
- authenticatorCancel, in 5.2.3
- authenticator data, in 5.3.1
- authenticatorData, in 4.2.2
- authenticator data claimed to have been used for the attestation, in 5.3.2
- authenticator data for the attestation, in 5.3.2
- authenticator extension, in 8
- authenticator extension input, in 6.3
- authenticator extension output, in 6.3
- Authenticator extension processing, in 8.5
- AuthenticatorExtensions, in 4.5.1
- authenticatorGetAssertion, in 5.2.2
- authenticatorMakeCredential, in 5.2.1
- AuthenticatorResponse, in 4.2
- AuthenticatorSelection, in 4.5
- AuthenticatorSelectionCriteria, in 4.5.2
- AuthenticatorSelectionList, in 9.4

- Authorization Gesture, in 3
- Base64url Encoding, in 2.1
- Basic Attestation, in 5.3.3
- Biometric Recognition, in 3
- Blob, in 4.8.4
- CBOR, in 2.1
- Ceremony, in 3
- challenge
  + dict-member for MakeCredentialOptions, in 4.5
  + dict-member for PublicKeyCredentialRequestOptions, in 4.6
  + dict-member for CollectedClientData, in 4.8.1
- Client, in 3
Terms defined by reference

* [CREDENTIAL-MANAGEMENT-1] defines the following terms:

  + CredentialCreationOptions

+ create()

* [ECMAScript] defines the following terms:

  + arraybuffer
  + internal slot
  + string

* [ENCODING] defines the following terms:

  + utf-8 encode

* [HTML] defines the following terms:

  + ascii serialization of an origin
  + dom manipulation task source
  + effective domain
  + global object

Terms defined by reference

* [CREDENTIAL-MANAGEMENT-1] defines the following terms:

+ Credential
+ CredentialCreationOptions
+ CredentialRequestOptions
+ CredentialsContainer
+ [CredentialContext](credential)
+ [credential]
+ [credential]
+ [credential]

+ type
+ xplat, in 4.4.4

+ 'xplat', in 4.4.4

+ 'xplat', in 4.4.4

+ 'xplat', in 4.4.4

+ Registration, in 3
+ registration extension, in 3
+ registration extension, in 3
+ registration extension, in 3
+ Registration, in 3
+ Relying Party, in 3
+ Relying Party Identifier, in 3
+ rp, in 4.3
+ [requestorID], in 4.4
+ [requestorID], in 4.4
+ [requestorID], in 4.4
+ [requestorID], in 4.4

+ response, in 4.1
+ response, in 4.1
+ response, in 4.1
+ response, in 4.1

+ requireResidentKey, in 4.5.2
+ requireResidentKey, in 4.5.2
+ requireResidentKey, in 4.5.2
+ requireResidentKey, in 4.5.2

+ [HTML] defines the following terms:

+ ascii serialization of an origin
+ dom manipulation task source
+ effective domain
+ global object

+ utf-8 encode
+ [HTML] defines the following terms:

+ utf-8 encode

+ utf-8 encode

+ utf-8 encode

+ utf-8 encode

+ utf-8 encode
* [WebIDL] defines the following terms:
  + recognized algorithm name
  + normalizing an algorithm

* [WebCryptoAPI] defines the following terms:
  + AlgorithmIdentifier
  + buffer source
  + buffer source
  + ConstraintError
  + DOMException
  + DOMString
  + NotAllowedError
  + NotFoundError
  + NotSupportedError
  + Promise
  + SameObject
  + SecurityContext
  + SecurityError
  + TypeError
  + USVString

* [HTTP] defines the following terms:
  + secure context
  + secure context
  + secure context
  + token binding id
  + token binding

* [secure-contexts] defines the following terms:
  + set
  + remove
  + ordered set

* [INFRA] defines the following terms:
  + origin
  + opaque origin
  + opaque origin
  + origin
  + domain
  + domain
  + domain
  + domain

* [URL] defines the following terms:
  + url serializer
4439 | {TPMv2-Part2} Trusted Platform Module Library, Part 2; Structures, URL: 
4440 | http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-
4441 | 2.0-Part-2-Structures-01.38.pdf

4442 | {TPMv2-Part3} Trusted Platform Module Library, Part 3; Commands, URL: 
4443 | http://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-
4444 | 2.0-Part-3-Commands-01.38.pdf

4445 | [UAFProtocol] R. Lindemann; et al. FIDO UAF Protocol Specification v1.0. FIDO 
4446 | Alliance Proposed Standard. URL: https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-ua
4447 | f-protocol-v1.0-ps-20141208.html

IDL Index

4448 | [SecureContext] interface PublicKeyCredential : Credential {
4449 |   required attribute ArrayBuffer rawId;
4450 |   readonly attribute AuthenticatorResponse response;
4451 |   readonly attribute AuthenticationExtensions clientExtensionResults;
4452 |
4453 | partial dictionary CredentialRequestOptions {
4454 |   PublicKeyCredentialRequestOptions? publicKey;
4455 |   };
4456 | partial dictionary CredentialCreationOptions {
4457 |   MakeCredentialOptions? publicKey;
4458 |   };
4459 |
4460 | [SecureContext] interface AuthenticatorResponse {
4461 |   readonly attribute ArrayBuffer clientDataJSON;
4462 |   };
4464 |   readonly attribute ArrayBuffer attestationObject;
4465 |   };
4466 |
4467 | dictionary PublicKeyCredentialParameters {
4468 |   required PublicKeyCredentialType type;
4469 |   required AlgorithmIdentifier algorithm;
4470 |   };
4471 |
4472 | dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
4473 |   DOMString displayName;
4474 |   };
4475 |
4476 | dictionary MakeCredentialOptions {
4477 |   required PublicKeyCredentialEntity rp;
4478 |   required PublicKeyCredentialUserEntity user;
4479 |   required BufferSource challenge;
4480 |   required sequence<PublicKeyCredentialParameters> parameters;
4481 |
4482 | [SecureContext] interface PublicKeyCredential : Credential {
4483 |   [SameObject] readonly attribute ArrayBuffer rawId;
4484 |   [SameObject] readonly attribute AuthenticatorResponse response;
4485 |   [SameObject] readonly attribute AuthenticationExtensions clientExtensionResults;
4486 |   };
4487 |
4488 | partial dictionary CredentialCreationOptions {
4489 |   MakePublicKeyCredentialOptions publicKey;
4490 |   };
4491 |
4492 | [SecureContext] interface PublicKeyCredentialRequestOptions {
4493 |   PublicKeyCredentialRequestOptions? publicKey;
4494 |   };
4495 |
4496 | [SecureContext] interface AuthenticatorResponse {
4497 |   [SameObject] readonly attribute ArrayBuffer clientDataJSON;
4498 |   };
4500 |   [SameObject] readonly attribute ArrayBuffer attestationObject;
4501 |   };
4503 |   [SameObject] readonly attribute ArrayBuffer authenticatorData;
4504 |   };
4505 |
4506 | dictionary PublicKeyCredentialParameters {
4507 |   required PublicKeyCredentialType type;
4508 |   required AlgorithmIdentifier algorithm;
4509 |   };
4510 |
4511 | dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
4512 |   DOMString displayName;
4513 |   };
4514 |
4515 | dictionary MakePublicKeyCredentialOptions {
4516 |   required PublicKeyCredentialEntity rp;
4517 |   required PublicKeyCredentialUserEntity user;
4518 |   required BufferSource challenge;
4519 |   required sequence<PublicKeyCredentialParameters> publicKeyCredParams;
4520 | }
typedef BufferSource AAGUID;
dictionary PublicKeyCredentialEntity {
  DOMString id;
  DOMString name;
  USVString icon;
};
dictionary PublicKeyCredentialEntity {
  DOMString id;
  DOMString name;
  USVString icon;
};
dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
  DOMString displayName;
};
dictionary PublicKeyCredentialEntity {
  DOMString id;
  DOMString name;
  USVString icon;
};
dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
  DOMString displayName;
};
dictionary AuthenticatorSelectionCriteria {
  enum Attachment {
    "platform",  // Platform attachment
    "xplat"  // Cross-platform attachment
  };
};
dictionary CollectedClientData {
  required DOMString challenge;
  required DOMString origin;
  required DOMString hashAlg;
  DOMString tokenBinding;
  AuthenticationExtensions clientExtensions;
  AuthenticationExtensions authenticatorExtensions;
};
dictionary PublicKeyCredentialDescriptor {
  required PublicKeyCredentialType type;
  required BufferSource id;
  sequence<Transport> transports;
};
dictionary PublicKeyCredentialDescriptor {
  required PublicKeyCredentialType type;
  required BufferSource id;
  sequence<Transport> transports;
};
dictionary AuthenticatorSelectionCriteria {
  enum Attachment {
    "platform",  // Platform attachment
    "xplat"  // Cross-platform attachment
  };
};
dictionary CollectedClientData {
  required DOMString challenge;
  required DOMString origin;
  required DOMString hashAlg;
  DOMString tokenBinding;
  AuthenticationExtensions clientExtensions;
  AuthenticationExtensions authenticatorExtensions;
};
dictionary PublicKeyCredentialDescriptor {
  required PublicKeyCredentialType type;
  required BufferSource id;
  sequence<Transport> transports;
};
dictionary AuthenticatorSelectionCriteria {
  enum Attachment {
    "platform",  // Platform attachment
    "xplat"  // Cross-platform attachment
  };
};
dictionary CollectedClientData {
  required DOMString challenge;
  required DOMString origin;
  required DOMString hashAlg;
  DOMString tokenBinding;
  AuthenticationExtensions clientExtensions;
  AuthenticationExtensions authenticatorExtensions;
};
dictionary PublicKeyCredentialDescriptor {
  required PublicKeyCredentialType type;
  required BufferSource id;
  sequence<Transport> transports;
};
dictionary AuthenticatorSelectionCriteria {
  enum Attachment {
    "platform",  // Platform attachment
    "xplat"  // Cross-platform attachment
  };
};
dictionary CollectedClientData {
  required DOMString challenge;
  required DOMString origin;
  required DOMString hashAlg;
  DOMString tokenBinding;
  AuthenticationExtensions clientExtensions;
  AuthenticationExtensions authenticatorExtensions;
};
dictionary PublicKeyCredentialDescriptor {
  required PublicKeyCredentialType type;
  required BufferSource id;
  sequence<Transport> transports;
};
dictionary AuthenticatorSelectionCriteria {
  enum Attachment {
    "platform",  // Platform attachment
    "xplat"  // Cross-platform attachment
  };
};
dictionary CollectedClientData {
  required DOMString challenge;
  required DOMString origin;
  required DOMString hashAlg;
  DOMString tokenBinding;
  AuthenticationExtensions clientExtensions;
  AuthenticationExtensions authenticatorExtensions;
};
dictionary PublicKeyCredentialDescriptor {
  required PublicKeyCredentialType type;
  required BufferSource id;
  sequence<Transport> transports;
};
dictionary AuthenticatorSelectionCriteria {
  enum Attachment {
    "platform",  // Platform attachment
    "xplat"  // Cross-platform attachment
  };
};
dictionary CollectedClientData {
  required DOMString challenge;
  required DOMString origin;
  required DOMString hashAlg;
  DOMString tokenBinding;
  AuthenticationExtensions clientExtensions;
  AuthenticationExtensions authenticatorExtensions;
};
dictionary PublicKeyCredentialDescriptor {
  required PublicKeyCredentialType type;
  required BufferSource id;
  sequence<Transport> transports;
};
dictionary AuthenticatorSelectionCriteria {
  enum Attachment {
    "platform",  // Platform attachment
    "xplat"  // Cross-platform attachment
  };
};
dictionary CollectedClientData {
  required DOMString challenge;
  required DOMString origin;
  required DOMString hashAlg;
  DOMString tokenBinding;
  AuthenticationExtensions clientExtensions;
  AuthenticationExtensions authenticatorExtensions;
};
dictionary PublicKeyCredentialDescriptor {
  required PublicKeyCredentialType type;
  required BufferSource id;
  sequence<Transport> transports;
};
6.2. Verifying an authentication assertion (2) (3) (4)

6.1. Registering a new credential (2) (3) (4) (5) (6) (7) (8) (9)

6. Relying Party Operations (2) (3) (4) (5)

Compromise (2) (3) (4) (5) (6)

5.3.1. Attestation data (2)

5.3.5.2. Attestation Certificate and Attestation Certificate CA Compromise (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13)

6.2. Verifying an authentication assertion (2) (3) (4)

7.4. Android Key Attestation Statement Format
5.3.5.2. Attestation Certificate and Attestation Certificate CA
5.3.4. Generating an Attestation Object
5.3.3. Attestation Types
5. WebAuthn Authenticator model
4.5.2. Authenticator Selection Criteria
4.5.2. Authenticator Selection Criteria
4.1.4. Use an existing credential - PublicKeyCredential's [[Create]](options) method
4.1.3. Create a new credential - PublicKeyCredential's [[DiscoverFromExternalSource]](options) method
4.4.1. Public Key Entity Description (dictionary)
4.4.1. Public Key Entity Description (dictionary)
4.1. PublicKeyCredential Interface
4.1. PublicKeyCredential Interface
4.4.3. Authenticator Selection Criteria (dictionary)
4.4.3. Authenticator Selection Criteria (dictionary)
4.6. Options for Assertion Generation (dictionary)
4.6. Options for Assertion Generation (dictionary)
4.5. Options for Credential Creation (dictionary)
4.5. Options for Credential Creation (dictionary)
4.4. Options for Credential Creation (dictionary)
4. Web Authentication API
3. Terminology (2) (3) (4) (5) (6)
2. Defining extensions (2)
1. Introduction (2) (3) (4) (5)
#relying-party-identifierReferenced in:
#rp-idReferenced in:
#public-key-credentialReferenced in:
#rp-idReferenced in:
#public-key-credentialReferenced in:
#rp-idReferenced in:
#public-key-credentialReferenced in:
#rp-idReferenced in:
#public-key-credentialReferenced in:
#rp-idReferenced in:
#public-key-credentialReferenced in:
#rp-idReferenced in:
#public-key-credentialReferenced in:
#rp-idReferenced in:
#public-key-credentialReferenced in:
#rp-idReferenced in:
#public-key-credentialReferenced in:
#rp-idReferenced in:
5. WebAuthn Authenticator model
5. WebAuthn Authenticator model
5.1. Authenticator data (2) (3) (4) (5) (6)
4. Web Authentication API
4. Web Authentication API
3. Terminology (2) (3) (4) (5)
1. Introduction (2) (3) (4) (5)
#public-key-credentialReferenced in:
#rp-idReferenced in:
#public-key-credentialReferenced in:
4.8.3. Credential Descriptor (dictionary)

4.1.4. Use an existing credential - PublicKeyCredential's

4.1.3. Create a new credential - PublicKeyCredential's

4.1. PublicKeyCredential Interface (2) (3) (4) (5) (6)

3. Terminology (2) (3) (4) (5) (6) (7) (8)

1. Introduction

#publickeycredentialReferenced in:

#web-authentication-apiReferenced in:

#webauthn-clientReferenced in:

#client-side-resident-credential-private-keyReferenced in:

#user-consentReferenced in:

#user-verificationReferenced in:

#test-of-user-presenceReferenced in:

#test-of-user-presenceReferenced in:

#user-consentReferenced in:

#user-verificationReferenced in:

#concept-user-presentReferenced in:

#concept-user-presentReferenced in:

#user-verifierReferenced in:

#user-verifierReferenced in:

#pubickeycredentialReferenced in:

#webauthn-clientReferenced in:

#web-authentication-apiReferenced in:

#publickeycredentialReferenced in:

#publickeycredentialReferenced in:

#publickeycredentialReferenced in:
### 4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse) (2)

- **4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse) (2)**

### 4.2. Authenticator Responses (interface AuthenticatorResponse) (2)

- **4.2. Authenticator Responses (interface AuthenticatorResponse) (2)**

### 4.1. PublicKeyCredential Interface

- **4.1. PublicKeyCredential Interface (2)**

### Dictionary Definitions

- **#dictdef-credentialrequestoptions**

### 4.1.3. Create a new credential - PublicKeyCredential's

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

### 4.1.4. Use an existing credential - PublicKeyCredential's

- **4.1.4. Use an existing credential - PublicKeyCredential's [[Create]](options) method**

### 6.2. Verifying a authentication assertion

- **6.2. Verifying a authentication assertion (2) (3)**

### 6.4. Client extension processing

- **6.4. Client extension processing (2) (3)**

### 6. Relying Party Operations

- **6. Relying Party Operations (2) (3)**
# PublicKeyCredential Interface

4.1.3. Create a new credential - PublicKeyCredential's

4.1.4. Use an existing credential - PublicKeyCredential's

4.4. Options for Credential Creation (dictionary PublicKeyCredentialParameters)

4.4.1. MakePublicKeyCredentialOptions) (2)

4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)

4.1.3. Create a new credential - PublicKeyCredential's

4.1.4. Use an existing credential to make an assertion - PublicKeyCredential's

4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)

4.1.3. Create a new credential - PublicKeyCredential's

4.1.4. Use an existing credential to make an assertion - PublicKeyCredential's

4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)

4.1.3. Create a new credential - PublicKeyCredential's

4.1.4. Use an existing credential to make an assertion - PublicKeyCredential's

4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)

4.1.3. Create a new credential - PublicKeyCredential's

4.1.4. Use an existing credential to make an assertion - PublicKeyCredential's

4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)

4.1.3. Create a new credential - PublicKeyCredential's

4.1.4. Use an existing credential to make an assertion - PublicKeyCredential's

4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)

4.1.3. Create a new credential - PublicKeyCredential's

4.1.4. Use an existing credential to make an assertion - PublicKeyCredential's

4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)

4.1.3. Create a new credential - PublicKeyCredential's

4.1.4. Use an existing credential to make an assertion - PublicKeyCredential's

4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)

4.1.3. Create a new credential - PublicKeyCredential's

4.1.4. Use an existing credential to make an assertion - PublicKeyCredential's

4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)

4.1.3. Create a new credential - PublicKeyCredential's

4.1.4. Use an existing credential to make an assertion - PublicKeyCredential's

4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)

4.1.3. Create a new credential - PublicKeyCredential's

4.1.4. Use an existing credential to make an assertion - PublicKeyCredential's

4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)

4.1.3. Create a new credential - PublicKeyCredential's

4.1.4. Use an existing credential to make an assertion - PublicKeyCredential's

4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)

4.1.3. Create a new credential - PublicKeyCredential's

4.1.4. Use an existing credential to make an assertion - PublicKeyCredential's

4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)

4.1.3. Create a new credential - PublicKeyCredential's

4.1.4. Use an existing credential to make an assertion - PublicKeyCredential's

4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)

4.1.3. Create a new credential - PublicKeyCredential's

4.1.4. Use an existing credential to make an assertion - PublicKeyCredential's

4.2.1. Information about Public Key Credential (interface AuthenticatorAttestationResponse)

4.2.2. Web Authentication Assertion (interface AuthenticatorAssertionResponse)
4.3. Parameters for Credential Generation (dictionary)
4.3. Parameters for Credential Generation (dictionary)

#dictdef-publickeycredentialuserentityReferenced in:
4.4. User Account Parameters for Credential Generation (dictionary)
MakePublicKeyCredentialUserEntity

#dictdef-makepublickeycredentialoptionsReferenced in:
4.4. Options for Credential Creation (dictionary)
MakePublicKeyCredentialOptions

#dictdef-makecredentialoptionsReferenced in:
4.2. CredentialCreationOptions Extension

#dom-makecredentialoptions-pReferenced in:
4.3. Create a new credential - PublicKeyCredential's

#dom-makecredentialoptions-parametersReferenced in:
4.3. Create a new credential - PublicKeyCredential's

#dom-makecredentialoptions-timeoutReferenced in:
4.3. Create a new credential - PublicKeyCredential's

#dom-makecredentialoptions-excludelistReferenced in:
4.3. Create a new credential - PublicKeyCredential's

#dom-makecredentialoptions-userReferenced in:
4.3. Create a new credential - PublicKeyCredential's

#dom-makecredentialoptions-rpReferenced in:
4.3. Create a new credential - PublicKeyCredential's

#dom-makecredentialoptionsReferenced in:
4.3. Create a new credential - PublicKeyCredential's

#dom-makepublickeycredentialoptionsReferenced in:
4.4. Options for Credential Creation (dictionary)
MakePublicKeyCredentialOptions

#dictdef-makepublickeycredentialoptionsReferenced in:
4.1.3. Create a new credential - PublicKeyCredential's

#dictdef-makepublickeycredentialoptions-userReferenced in:
4.1.3. Create a new credential - PublicKeyCredential's

#dictdef-makepublickeycredentialoptions-pReferenced in:
4.1.3. Create a new credential - PublicKeyCredential's

#dictdef-makepublickeycredentialoptions-parametersReferenced in:
4.1.3. Create a new credential - PublicKeyCredential's

#dictdef-makepublickeycredentialoptions-timeoutReferenced in:
4.1.3. Create a new credential - PublicKeyCredential's

#dictdef-makepublickeycredentialoptions-excludelistReferenced in:
4.1.3. Create a new credential - PublicKeyCredential's

#dictdef-makepublickeycredentialoptions-parametersReferenced in:
4.1.3. Create a new credential - PublicKeyCredential's
* 4.5.2. Authenticator Selection Criteria

* 5.2.1. The authenticatorMakeCredential operation

* dom-makecredentialoptions-extensionsReferenced in:
  * 4.1.3. Create a new credential - PublicKeyCredential's
    * [[Create]](options) method (2)
  * 4.4. Options for Credential Creation (dictionary PublicKeyCredentialOptions) (2)
  * 4.4.1. Public Key Entity Description (dictionary PublicKeyCredentialUserEntity) (2)
  * 4.4.2. User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity) (2)

* 4.4. Options for Credential Creation (dictionary PublicKeyCredentialOptions) (2)

* 4.4.1. Public Key Entity Description (dictionary PublicKeyCredentialUserEntity) (2)

* 4.4.2. User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity) (2)

* 4.4.3. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria) (2)

* 4.4.2. User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity) (2)

* 4.4.3. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria) (2)

* 4.4.3. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria) (2)

* 4.4.2. User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity) (2)

* 4.4.3. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria) (2)

* 4.4.2. User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity) (2)

* 4.4.3. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria) (2)

* 4.4.2. User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity) (2)

* 4.4.3. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria) (2)

* 4.4.2. User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity) (2)

* 4.4.3. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria) (2)

* 4.4.2. User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity) (2)

* 4.4.3. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria) (2)

* 4.4.2. User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity) (2)

* 4.4.3. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria) (2)

* 4.4.2. User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity) (2)

* 4.4.3. Authenticator Selection Criteria (dictionary AuthenticatorSelectionCriteria) (2)
4.8.1. Client data used in WebAuthn signatures (dictionary PublicKeyCredential::[[DiscoverFromExternalSource]](options) method)

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

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

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

4.5. Options for Credential Creation (dictionary MakeCredentialOptions) (2)

4.6. Options for Assertion Generation (dictionary)

4.7.1. Client data used in WebAuthn signatures (dictionary CollectedClientData) (2)

4.2. Authenticator Responses (interface AuthenticatorResponse)

5.1. Authenticator data (2) (3) (4)

5.2. Verifying an authentication assertion

8. WebAuthn Extensions

8.6. Example Extension

5.1. Registering a new credential

5.2. Verifying an authentication assertion

8.6. Example Extension

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

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

4.5. Options for Assertion Generation (dictionary MakePublicKeyCredentialOptions) (2)

4.5.1. Client data used in WebAuthn signatures (dictionary CollectedClientData) (2)

5.1. Authenticator data (2)

5.2. Verifying an authentication assertion

8. WebAuthn Extensions

8.4. Client extension processing

8.6. Example Extension

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

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

4.7.1. Client data used in WebAuthn signatures (dictionary CollectedClientData) (2)

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

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

4.7.1. Client data used in WebAuthn signatures (dictionary CollectedClientData) (2)

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

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

4.8.1. Client data used in WebAuthn signatures (dictionary)

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

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

4.8.1. Client data used in WebAuthn signatures (dictionary)

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

4.7.1. Client data used in WebAuthn signatures (dictionary CollectedClientData) (2)

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

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

4.8.1. Client data used in WebAuthn signatures (dictionary)

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

4.7.1. Client data used in WebAuthn signatures (dictionary CollectedClientData) (2)
5.3.4. Generating an Attestation Object

5.3.2. Attestation Statement Formats (2)

5.2.2. The authenticatorGetAssertion operation (2) (3)

5.2.1. The authenticatorMakeCredential operation (2)

4.2.1. Information about Public Key Credential (interface PublicKeyCredential)

#collectedclientdata-tokenBindingReferenced in:

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

4.1.4. Using an existing credential - PublicKeyCredential's [[DiscoverFromExternalSource](options) method

4.6.1. Client data used in WebAuthn signatures (dictionary)

CollectedClientData

4.1. Registering a new credential

4.2. Verifying an authentication assertion

#collectedclientdata-clientextensionsReferenced in:

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

4.1.4. Using an existing credential - PublicKeyCredential's [[DiscoverFromExternalSource](options) method

4.6.1. Client data used in WebAuthn signatures (dictionary)

CollectedClientData

4.1. Registering a new credential

4.2. Verifying an authentication assertion

8.4. Client extension processing

#collectedclientdata-authenticatorextensionsReferenced in:

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

4.1.4. Using an existing credential - PublicKeyCredential's [[DiscoverFromExternalSource](options) method

4.6.1. Client data used in WebAuthn signatures (dictionary)

CollectedClientData

4.1. Registering a new credential

4.2. Verifying an authentication assertion

#collectedclientdata-json-serialized-client-dataReferenced in:

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

4.1.4. Using an existing credential - PublicKeyCredential's [[DiscoverFromExternalSource](options) method

4.6.1. Client data used in WebAuthn signatures (dictionary)

CollectedClientData

4.1. Registering a new credential

4.2. Verifying an authentication assertion

#collectedclientdata-hash-of-the-serialized-client-dataReferenced in:

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

4.1.4. Using an existing credential - PublicKeyCredential's [[DiscoverFromExternalSource](options) method

4.6.1. Client data used in WebAuthn signatures (dictionary)

CollectedClientData

5. WebAuthn Authenticator model

5.2.1. The authenticatorMakeCredential operation (2)

5.2.2. The authenticatorGetAssertion operation (2) (3)

5.3.2. Attestation Statement Formats (2)

5.3.4. Generating an Attestation Object

6.1. Registering a new credential

582
* 4.8.4. Credential Transport enumeration (enum ExternalTransport)

#dom-transport-ble
Referenced in:
* 4.8.4. Credential Transport enumeration (enum ExternalTransport)

#dom-transport-usb
Referenced in:
* 4.8.3. Credential Descriptor (dictionary

#enumdef-transport
Referenced in:
* 4.7.4. Authenticator Transport enumeration (enum AuthenticatorTransport)

#enumdef-ble
Referenced in:
* 4.8.4. Credential Transport enumeration (enum ExternalTransport)

#enumdef-usb
Referenced in:
* 4.8.4. Credential Transport enumeration (enum ExternalTransport)

#enumdef-nfc
Referenced in:
* 4.8.4. Credential Transport enumeration (enum ExternalTransport)

#enumdef-ble
Referenced in:
* 4.8.4. Credential Transport enumeration (enum ExternalTransport)

#dom-ble
type:
* 4.8.4. Credential Transport enumeration (enum ExternalTransport)

#dom-ble
Referenced in:
* 4.4. Authenticator Transport enumeration (enum AuthenticatorTransport)

#dom-ble
Referenced in:
* 4.4. Authenticator Transport enumeration (enum AuthenticatorTransport)

#dom-ble
Referenced in:
* 4.4. Authenticator Transport enumeration (enum AuthenticatorTransport)

#dom-ble
Referenced in:
* 4.4. Authenticator Transport enumeration (enum AuthenticatorTransport)

#dom-ble
Referenced in:
* 4.4. Authenticator Transport enumeration (enum AuthenticatorTransport)
#attestation-statement-format
Referenced in:
#authenticatormakecredential
Referenced in:
#assertion-signature
Referenced in:
#attestation-signature
Referenced in:
#authenticatorcancel
Referenced in:
#assertion-signature
Referenced in:
#assertion-signature
Referenced in:
#authenticatorcancel
Referenced in:
#assertion-signature
Referenced in:
#authenticatorcancel
Referenced in:
#assertion-signature
Referenced in:
#authenticatorcancel
Referenced in:
#assertion-signature
Referenced in:
#authenticatorcancel
Referenced in:
#assertion-signature
Referenced in:
#authenticatorcancel
Referenced in:
#assertion-signature
Referenced in:
#authenticatorcancel
Referenced in:
#assertion-signature
Referenced in:
#authenticatorcancel
Referenced in:
#assertion-signature
Referenced in:
#authenticatorcancel
Referenced in:
#assertion-signature
Referenced in:
#authenticatorcancel
Referenced in:
#assertion-signature
Referenced in:
#authenticatorcancel
Referenced in:
#assertion-signature
Referenced in:
#authenticatorcancel
Referenced in:
#assertion-signature
Referenced in:
#authenticatorcancel
Referenced in:
#assertion-signature
Referenced in:
#authenticatorcancel
Referenced in:
#assertion-signature
Referenced in:
#authenticatorcancel
Referenced in:
#assertion-signature
Referenced in:
#authenticatorcancel
Referenced in:
#assertion-signature
Referenced in:
#authenticatorcancel
Referenced in:
#assertion-signature
Referenced in:
#authenticatorcancel
Referenced in:
#assertion-signature
Referenced in:
* 3. Terminology

* 4.2.1. Information about Public Key Credential (interface)

#attestation-type

* 5.3.4. Generating an Attestation Object (2)

#authenticator-data

* 5.1. Authenticator data (2) (3) (4) (5) (6) (7)

* 5.2. The authenticatorCreateCredential operation

* 5.2.1. The authenticatorMakeCredential operation

* 5.2.2. The authenticatorGetAssertion operation

* 5.3. Credential Attestation (2)

* 5.3.3. Attestation Types

* 6.1. Registering a new credential (2)

* 7.3. TPM Attestation Statement Format

* 7.4. Android Key Attestation Statement Format

#authenticator-data-for-the-attestation

* 7.2. Packet Attestation Statement Format

* 7.3. TPM Attestation Statement Format

* 7.4. Android Key Attestation Statement Format (2)

* 7.5. Android SafetyNet Attestation Statement Format

* 7.6. FIDO U2F Attestation Statement Format

#authenticator-data-claimed-to-have-been-used-for-the-attestation

* 7.2. Packed Attestation Statement Format

* 7.3. TPM Attestation Statement Format

* 7.4. Android Key Attestation Statement Format (2)

* 7.5. Android SafetyNet Attestation Statement Format

* 7.6. FIDO U2F Attestation Statement Format

#signing-procedure

* 5.3.2. Attestation Statement Formats

#attestation-statement

* 5.1. Authenticator data (2) (3) (4) (5) (6) (7)

* 5.2. The authenticatorGetAssertion operation

* 5.3. Attestation (2)

* 5.3.3. Attestation Types

* 6.1. Registering a new credential (2)

* 7.3. TPM Attestation Statement Format

* 7.4. Android Key Attestation Statement Format

#authenticator-data-for-the-attestation

* 7.2. Packed Attestation Statement Format

* 7.3. TPM Attestation Statement Format

* 7.4. Android Key Attestation Statement Format (2)

* 7.5. Android SafetyNet Attestation Statement Format

* 7.6. FIDO U2F Attestation Statement Format

#authenticator-data-claimed-to-have-been-used-for-the-attestation

* 7.2. Packed Attestation Statement Format

* 7.3. TPM Attestation Statement Format

* 7.4. Android Key Attestation Statement Format (2)

* 7.5. Android SafetyNet Attestation Statement Format

* 7.6. FIDO U2F Attestation Statement Format

#basic-attestation

* 5.3.5.1. Privacy

* 5.3.5.2. Attestation Certificate and Attestation Certificate CA Compromise

* 5.3.5.2. Attestation Certificate and Attestation Certificate CA Compromise

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

* 5.3. Credential Attestation

* 5.3.1. Privacy

* 5.3.2. Attestation Statement Formats

* 5.3.3. Attestation Types

* 5.3.2. Attestation Certificate and Attestation Certificate CA Compromise

* 6.1. Registering a new credential (2) (3)
8.4. Client extension processing

5.3.5.1. Privacy

* elliptic-curve-based-direct-anonymous-attestation

#client-extensionReferenced in:
* 8.4. Client extension processing

9.1. FIDO AppId Extension (appid)

* 5.3.5.1. Privacy

#client-extensionReferenced in:
* 8.4. Client extension processing

7.2. Packed Attestation Statement Format (2)

10.2. WebAuthn Extension Identifier Registrations (2) (3)

5.3.3. Attestation Types

7.3. TPM Attestation Statement Format (2)

10.2. WebAuthn Extension Identifier Registrations (2) (3) (4) (5)

5.3.2. Attestation Statement Formats

#registration-extensionReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's [[Create](options)] method

5.3.3. Attestation Certificate and Attestation Certificate CA Compromise

5.3.2. Attestation Statement Formats

10.2. WebAuthn Extension Identifier Registrations (2) (3) (4) (5) (6)

5.3.1. Privacy

6.1. Registering a new credential

7.2. Packed Attestation Statement Format

6.1. Registering a new credential

7.3. TPM Attestation Statement Format (2)

#authentication-extensionReferenced in:
* 4.1.4. Use an existing credential to make an assertion - PublicKeyCredential's [[DiscoverFromExternalSource](options)] method

8. WebAuthn Extensions (2) (3) (4) (5) (6)

9.2. Simple Transaction Authorization Extension (txAuthSimple)

9.1. FIDO AppId Extension (appid)

9.2. Simple Transaction Authorization Extension (txAuthSimple)

9.1. FIDO AppId Extension (appid)

9.3. Generic Transaction Authorization Extension (txAuthGeneric)

9.4. Authenticator Selection Extension (authnSel)

9.5. Supported Extensions Extension (exts)

9.8. User Verification Index Extension (uv)

6.1. Registering a new credential

7.2. Packed Attestation Statement Format (2) (3)

6.1. Registering a new credential

7.2. Packed Attestation Statement Format

6.1. Registering a new credential

7.2. Packed Attestation Statement Format (2) (3)

6.1. Registering a new credential

7.2. Packed Attestation Statement Format (2)

6.1. Registering a new credential

7.2. Packed Attestation Statement Format

6.1. Registering a new credential

7.2. Packed Attestation Statement Format
#authenticator-extensionReferenced in:
* 4.1.3. Create a new credential - PublicKeyCredential's [[Create](options)] method (2)
* 4.1.4. Use an existing credential - PublicKeyCredential's [[DiscoverFromExternalSource](options)] method (2)

#extension-identifierReferenced in:
* 8. WebAuthn Extensions (2) (3) (4)
* 8.2. Defining extensions (2) (3) (4)
* 8.3. Extending request parameters (2) (3) (4)
* 8.5. Authenticator extension processing (2) (3) (4)

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

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

#extension-identifierReferenced in:
* 8. WebAuthn Extensions (2) (3) (4)
* 8.2. Defining extensions (2) (3) (4)
* 8.3. Extending request parameters (2) (3) (4)
* 8.5. Authenticator extension processing (2) (3) (4)

#authenticator-extension_inputReferenced in:
* 8. WebAuthn Extensions (2) (3) (4)
* 8.2. Defining extensions (2) (3) (4)
* 8.3. Extending request parameters (2) (3) (4)
* 8.4. Client extension processing (2) (3) (4)
* 8.5. Authenticator extension processing (2) (3) (4)

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

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

#extension-identifierReferenced in:
* 8. WebAuthn Extensions (2) (3)
* 8.2. Defining extensions (2) (3)
* 8.4. Client extension processing (2) (3) (4)
* 8.5. Authenticator extension processing (2) (3) (4)

#extension-identifierReferenced in:
* 8. WebAuthn Extensions (2) (3) (4) (5)
* 8.2. Defining extensions (2) (3) (4)
* 8.3. Extending request parameters (2) (3) (4)
* 8.4. Client extension processing (2) (3) (4)
* 8.5. Authenticator extension processing (2) (3) (4)

#client-extension-processingReferenced in:
* 4.1. PublicKeyCredential Interface
* 4.1.3. Create a new credential - PublicKeyCredential's [[Create](options)] method (2)
* 4.1.4. Use an existing credential to make an assertion - PublicKeyCredential's [[DiscoverFromExternalSource](options)] method (2)

#authenticator-extension-processingReferenced in:
* 8. WebAuthn Extensions (2) (3) (4) (5)
* 8.2. Defining extensions (2) (3) (4)
* 8.4. Client extension processing (2) (3) (4)
* 8.5. Authenticator extension processing (2) (3) (4)

#client-extension-processingReferenced in:
* 4.1. PublicKeyCredential Interface
* 4.1.3. Create a new credential - PublicKeyCredential's [[Create](options)] method (2)
* 4.1.4. Use an existing credential to make an assertion - PublicKeyCredential's [[DiscoverFromExternalSource](options)] method (2)

#client-extension-processingReferenced in:
* 8. WebAuthn Extensions (2) (3) (4) (5)
* 8.2. Defining extensions (2) (3) (4)
* 8.4. Client extension processing (2) (3) (4)
* 8.5. Authenticator extension processing (2) (3) (4)
8.5. Authenticator extension processing

authenticate-extension-output

5.1. Authenticator data

8.2. Defining extensions (2) (3)

8.3. Client extension processing

8.5. Authenticator extension processing

8.6. Example Extension

5.5. Supported Extensions Extension (exts)

9.6. User Verification Index Extension (uvi)

9.7. Location Extension (loc)

9.8. User Verification Method Extension (uvm)

typedef-def-authenticatorselectionlist

typedef-def-aaguid

9.4. Authenticator Selection Extension (authnSel)

9.7. Location Extension (loc)

9.8. User Verification Method Extension (uvm)

typedef-def-authenticatorselectionlist

typedef-def-aaguid

9.4. Authenticator Selection Extension (authnSel)