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

partial dictionary CredentialCreationOptions {
    PublicKeyCredentialCreationOptions      publicKey;
};

partial dictionary CredentialRequestOptions {
    PublicKeyCredentialRequestOptions      publicKey;
};

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

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

[SecureContext, Exposed=Window]
interface AuthenticatorAttestationResponse : AuthenticatorResponse {
    [SameObject] readonly attribute ArrayBuffer      attestationObject;
    sequence<DOMString>                              getTransports();
    ArrayBuffer                                      getAuthenticatorData();
```

```
    ArrayBuffer?                                getPublicKey();
    COSEAlgorithmIdentifier                     getPublicKeyAlgorithm();
};

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

dictionary PublicKeyCredentialParameters {
    required DOMString                    type;
    required COSEAlgorithmIdentifier      alg;
};

dictionary PublicKeyCredentialCreationOptions {
    required PublicKeyCredentialRpEntity         rp;
    required PublicKeyCredentialUserEntity       user;

    required BufferSource                            challenge;
    required sequence<PublicKeyCredentialParameters>  pubKeyCredParams;

    unsigned long                               timeout;
    sequence<PublicKeyCredentialDescriptor>     excludeCredentials = [];
    AuthenticatorSelectionCriteria              authenticatorSelection;
    DOMString                                   attestation = "none";
    AuthenticationExtensionsClientInputs        extensions;
};

dictionary PublicKeyCredentialEntity {
```

```
    required DOMString    name;
};

dictionary PublicKeyCredentialRpEntity : PublicKeyCredentialEntity {
    DOMString      id;
};

dictionary PublicKeyCredentialUserEntity : PublicKeyCredentialEntity {
    required BufferSource   id;
    required DOMString      displayName;
};

dictionary AuthenticatorSelectionCriteria {
    DOMString                   authenticatorAttachment;
    DOMString                   residentKey;
    boolean                     requireResidentKey = false;
    DOMString                   userVerification = "preferred";
};

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

enum ResidentKeyRequirement {
    "discouraged",
    "preferred",
    "required"
};

enum AttestationConveyancePreference {
```

```
    "none",
    "indirect",
    "direct",
    "enterprise"
};

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

dictionary AuthenticationExtensionsClientInputs {
};

dictionary AuthenticationExtensionsClientOutputs {
};

dictionary CollectedClientData {
    required DOMString          type;
    required DOMString          challenge;
    required DOMString          origin;
    boolean                     crossOrigin;
    TokenBinding                tokenBinding;
};

dictionary TokenBinding {
    required DOMString status;
```

```
    DOMString id;
};

enum TokenBindingStatus { "present", "supported" };

enum PublicKeyCredentialType {
    "public-key"
};

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

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

typedef long COSEAlgorithmIdentifier;

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

partial dictionary AuthenticationExtensionsClientInputs {
```

```
  USVString appid;
};

partial dictionary AuthenticationExtensionsClientOutputs {
  boolean appid;
};

partial dictionary AuthenticationExtensionsClientInputs {
  USVString appidExclude;
};

partial dictionary AuthenticationExtensionsClientOutputs {
  boolean appidExclude;
};

partial dictionary AuthenticationExtensionsClientInputs {
  boolean uvm;
};

typedef sequence<unsigned long> UvmEntry;
typedef sequence<UvmEntry> UvmEntries;

partial dictionary AuthenticationExtensionsClientOutputs {
  UvmEntries uvm;
};

partial dictionary AuthenticationExtensionsClientInputs {
    boolean credProps;
};

dictionary CredentialPropertiesOutput {
```

```
    boolean rk;
};

partial dictionary AuthenticationExtensionsClientOutputs {
    CredentialPropertiesOutput credProps;
};

partial dictionary AuthenticationExtensionsClientInputs {
    AuthenticationExtensionsLargeBlobInputs largeBlob;
};

enum LargeBlobSupport {
  "required",
  "preferred",
};

dictionary AuthenticationExtensionsLargeBlobInputs {
    DOMString support;
    boolean read;
    BufferSource write;
};

partial dictionary AuthenticationExtensionsClientOutputs {
    AuthenticationExtensionsLargeBlobOutputs largeBlob;
};

dictionary AuthenticationExtensionsLargeBlobOutputs {
    boolean supported;
    ArrayBuffer blob;
```

```
    boolean written;
};
```

Web Authentication: An API for accessing Public Key Credentials Level 2

W3C Recommendation, 8 April 2021

This version:

https://www.w3.org/TR/2021/REC-webauthn-2-20210408/

Latest published version:

https://www.w3.org/TR/webauthn-2/

Editor's Draft:

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

Previous Versions:

https://www.w3.org/TR/2021/PR-webauthn-2-20210225/

https://www.w3.org/TR/2020/CR-webauthn-2-20201222/

https://www.w3.org/TR/2020/WD-webauthn-2-20201216/

https://www.w3.org/TR/2020/WD-webauthn-2-20201116/

https://www.w3.org/TR/2020/WD-webauthn-2-20200730/

https://www.w3.org/TR/2019/WD-webauthn-2-20191126/

https://www.w3.org/TR/2019/WD-webauthn-2-20190604/

https://www.w3.org/TR/2019/REC-webauthn-1-20190304/

Implementation Report:

https://www.w3.org/2020/12/webauthn-report.html

Issue Tracking:

GitHub

Editors:

Jeff Hodges (Google)

J.C. Jones (Mozilla)

Michael B. Jones (Microsoft)

Akshay Kumar (Microsoft)

Emil Lundberg (Yubico)

Former Editors:

Dirk Balfanz (Google)

Vijay Bharadwaj (Microsoft)

Arnar Birgisson (Google)

Alexei Czeskis (Google)

Hubert Le Van Gong (PayPal)

Angelo Liao (Microsoft)

Rolf Lindemann (Nok Nok Labs)

Contributors:

John Bradley (Yubico)

Christiaan Brand (Google)

Adam Langley (Google)

Giridhar Mandyam (Qualcomm)

Nina Satragno (Google)

Nick Steele (Gemini)

Jiewen Tan (Apple)

Shane Weeden (IBM)
Mike West (Google)
Jeffrey Yasskin (Google)

Tests:

web-platform-tests webauthn/ (ongoing work)

Please check the **errata** for any errors or issues reported since publication.

Abstract

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

Status of this document

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

This document was published by the Web Authentication Working Group as a Recommendation.

Feedback and comments on this specification are welcome. Please use Github issues. Discussions may also be found in the public-webauthn@w3.org archives.

A W3C Recommendation is a specification that, after extensive consensus-building, has received the endorsement of the W3C and its Members. W3C recommends the wide deployment of this specification as a standard for the Web.

This document has been reviewed by W3C Members, by software developers, and by other W3C groups and interested parties, and is endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited from another document. W3C's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

This document was produced by a group operating under the 1 August 2017 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

This document is governed by the 15 September 2020 W3C Process Document.

Table of Contents

§ 1. Introduction

This section is not normative.

This specification defines an API enabling the creation and use of strong, attested, scoped, public key-based credentials by web applications, for the purpose of strongly authenticating users. A public key credential is created and stored by a *WebAuthn Authenticator* at the behest of a *WebAuthn 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 scoped to a Relying Party with the present user's account (the account might already exist or might be created at this time). The second is Authentication, where the Relying Party is presented with an *Authentication Assertion* proving the presence and consent of the user who registered the public key credential. Functionally, the Web Authentication API comprises a `PublicKeyCredential` which extends the Credential Management API [CREDENTIAL-MANAGEMENT-1], and infrastructure which allows those credentials to be used with `navigator.credentials.create()` and `navigator.credentials.get()`. The former is used during Registration, and the latter during Authentication.

Broadly, compliant authenticators protect public key credentials, and interact with user agents to implement the Web Authentication API. Implementing compliant authenticators is possible in software executing (a) on a general-purpose computing device, (b) on an on-device Secure Execution Environment, Trusted Platform Module (TPM), or a Secure Element (SE), or (c) off device. Authenticators being implemented on device are called platform authenticators. Authenticators being implemented off device (roaming authenticators) can be accessed over a transport such as Universal Serial Bus (USB), Bluetooth Low Energy (BLE), or Near Field Communications (NFC).

§ 1.1. Specification Roadmap

While many W3C specifications are directed primarily to user agent developers and also to web application developers (i.e., "Web authors"), the nature of Web Authentication requires that this specification be correctly used by multiple audiences, as described below.

All audiences ought to begin with § 1.2 Use Cases, § 1.3 Sample API Usage Scenarios, and § 4 Terminology, and should also refer to [WebAuthnAPIGuide] for an overall tutorial. Beyond that, the intended audiences for this document are the following main groups:

- Relying Party web application developers, especially those responsible for Relying Party web application login flows, account recovery flows, user account database content, etc.
- Web framework developers
    - The above two audiences should in particular refer to § 7 WebAuthn Relying Party Operations. The introduction to § 5 Web Authentication API may be helpful, though readers should realize that the § 5 Web Authentication API section is targeted specifically at user agent developers, not web application developers. Additionally, if they intend to verify authenticator attestations, then § 6.5 Attestation and § 8 Defined Attestation Statement Formats will also be relevant. § 9 WebAuthn Extensions, and § 10 Defined Extensions will be of interest if they wish to make use of extensions. Finally, they should read § 13.4 Security considerations for Relying Parties and § 14.6 Privacy considerations for Relying Parties and consider which challenges apply to their application and users.
- User agent developers
- OS platform developers, responsible for OS platform API design and implementation in regards to platform-specific authenticator APIs, platform WebAuthn Client instantiation, etc.
    - The above two audiences should read § 5 Web Authentication API very carefully, along with § 9 WebAuthn Extensions if they intend to support extensions. They should also carefully read § 14.5 Privacy considerations for clients.
- Authenticator developers. These readers will want to pay particular attention to § 6 WebAuthn Authenticator Model, § 8 Defined Attestation Statement Formats, § 9 WebAuthn Extensions, and § 10 Defined Extensions. They should also carefully read § 13.3 Security considerations for authenticators and § 14.4 Privacy considerations for authenticators.

Note: Along with the Web Authentication API itself, this specification defines a request-response *cryptographic protocol*—the ***WebAuthn/FIDO2 protocol***—between a WebAuthn Relying Party server and an authenticator, where the Relying Party's request consists of a challenge and other input data supplied by the Relying Party and sent to the authenticator. The request is conveyed via the combination of HTTPS, the Relying Party web application, the WebAuthn API, and the platform-specific communications channel between the user agent and the authenticator. The authenticator replies with a digitally signed authenticator data message and other output data, which is conveyed back to the Relying Party server via the same path in reverse. Protocol details vary according to whether an authentication or registration operation is invoked by the Relying Party. See also Figure 1 and Figure 2.

It is important for Web Authentication deployments' end-to-end security that the role of each component—the Relying Party server, the client, and the authenticator— as well as § 13 Security Considerations and § 14 Privacy Considerations, are understood *by all audiences*.

§ 1.2. 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 § 1.3 Sample API Usage Scenarios.

§ 1.2.1. 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."

§ 1.2.2. Authentication

- On a laptop or desktop:
  - User pairs their phone with the laptop or desktop via Bluetooth.
  - User navigates to example.com in a browser and initiates signing in.
  - User gets a message from the browser, "Please complete this action on your phone."
- Next, on their phone:
  - User sees a discrete prompt or notification, "Sign in to example.com."
  - User selects this prompt / notification.
  - User is shown a list of their example.com identities, e.g., "Sign in as Mohamed / Sign in as 张三".
  - User picks an identity, is prompted for an authorization gesture (PIN, biometric, etc.) and provides this.
- Now, back on the laptop:
  - Web page shows that the selected user is signed in, and navigates to the signed-in page.

§ 1.2.3. New Device Registration

This use case scenario illustrates how a Relying Party can leverage a combination of a roaming authenticator (e.g., a USB security key fob) and a platform authenticator (e.g., a built-in fingerprint sensor) such that the user has:

- a "primary" roaming authenticator that they use to authenticate on new-to-them client devices (e.g., laptops, desktops) or on such client devices that lack a platform authenticator, and
- a low-friction means to strongly re-authenticate on client devices having platform authenticators.

> Note: This approach of registering multiple authenticators for an account is also useful in account recovery use cases.

- First, on a desktop computer (lacking a platform authenticator):
    - 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.
    - User navigates to account security settings and selects "Register security key".
    - Website prompts the user to plug in a USB security key fob; the user does.
    - The USB security key blinks to indicate the user should press the button on it; the user does.
    - Website shows message, "Registration complete."

    > Note: Since this computer lacks a platform authenticator, the website may require the user to present their USB security key from time to time or each time the user interacts with the website. This is at the website's discretion.

- Later, on their laptop (which features a platform authenticator):
    - User navigates to example.com in a browser and initiates signing in.
    - Website prompts the user to plug in their USB security key.
    - User plugs in the previously registered USB security key and presses the button.
    - Website shows that the user is signed in, and navigates to the signed-in page.
    - Website prompts, "Do you want to register this computer with example.com?"
    - User agrees.

    - Laptop prompts the user for a previously configured authorization gesture (PIN, biometric, etc.); the user provides this.
    - Website shows message, "Registration complete."
    - User signs out.
- Later, again on their laptop:
    - User navigates to example.com in a browser and initiates signing in.
    - Website shows message, "Please follow your computer's prompts to complete sign in."
    - Laptop prompts the user for an authorization gesture (PIN, biometric, etc.); the user provides this.
    - Website shows that the user is signed in, and navigates to the signed-in page.

§ 1.2.4. Other Use Cases and Configurations

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

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

§ 1.3. Sample API Usage 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 a first-factor roaming authenticator with its own display. One example of such an authenticator would be a smart phone. Other authenticator types are also supported by this API, subject to implementation by the client platform. For instance, this flow also works without modification for the case of an authenticator that is embedded in the client device. 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.

§ 1.3.1. Registration

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

1. The user visits example.com, which serves up a script. At this point, the user may already be logged in using a legacy username and password, or additional authenticator, or other means acceptable to the Relying Party. Or the user may be in the process of creating a new account.
2. The Relying Party script runs the code snippet below.
3. The client platform searches for and locates the authenticator.
4. The client connects to the authenticator, performing any pairing actions if necessary.
5. The authenticator shows appropriate UI for the user to provide a biometric or other authorization gesture.

6. The authenticator returns a response to the client, 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:

```
EXAMPLE 1
if (!window.PublicKeyCredential) { /* Client not capable. Handle error. */ }

var publicKey = {
  // The challenge is produced by the server; see the Security Considerations
  challenge: new Uint8Array([21,31,105 /* 29 more random bytes generated by the server */]

  // Relying Party:
  rp: {
    name: "ACME Corporation"
  },

  // User:
  user: {
    id: Uint8Array.from(window.atob("MIIBkzCCATigAwIBAjCCAZMwggE4oAMCAQIwggGTMII="), c=>c.
    name: "alex.mueller@example.com",
    displayName: "Alex Müller",
  },

  // This Relying Party will accept either an ES256 or RS256 credential, but
  // prefers an ES256 credential.
  pubKeyCredParams: [
    {
      type: "public-key",
      alg: -7 // "ES256" as registered in the IANA COSE Algorithms registry
    },
    {
      type: "public-key",
```

```
      alg: -257 // Value registered by this specification for "RS256"
    }
  ],

  authenticatorSelection: {
    // Try to use UV if possible. This is also the default.
    userVerification: "preferred"
  },

  timeout: 360000,  // 6 minutes
  excludeCredentials: [
    // Don't re-register any authenticator that has one of these credentials
    {"id": Uint8Array.from(window.atob("ufJWp8YGlibm1Kd9XQBWN1WAw2jy5In2Xhon9HAqcXE="), c=
    {"id": Uint8Array.from(window.atob("E/e1dhZc++mIsz4f9hb6NifAzJpF1V4mEtRlIPBiWdY="), c=
  ],

  // Make excludeCredentials check backwards compatible with credentials registered with U
  extensions: {"appidExclude": "https://acme.example.com"}
};

// Note: The following call will cause the authenticator to display UI.
navigator.credentials.create({ publicKey })
  .then(function (newCredentialInfo) {
    // Send new credential info to server for verification and registration.
  }).catch(function (err) {
    // No acceptable authenticator or user refused consent. Handle appropriately.
  });
```

§ 1.3.2. Registration Specifically with User-Verifying Platform Authenticator

This is an example flow for when the WebAuthn Relying Party is specifically interested in creating a public key credential with a user-verifying platform authenticator.

1. The user visits example.com and clicks on the login button, which redirects the user to login.example.com.
2. The user enters a username and password to log in. After successful login, the user is redirected back to example.com.
3. The Relying Party script runs the code snippet below.
    1. The user agent checks if a user-verifying platform authenticator is available. If not, terminate this flow.
    2. The Relying Party asks the user if they want to create a credential with it. If not, terminate this flow.
    3. The user agent and/or operating system shows appropriate UI and guides the user in creating a credential using one of the available platform authenticators.

4. Upon successful credential creation, the Relying Party script conveys the new credential to the server.

```
EXAMPLE 2
if (!window.PublicKeyCredential) { /* Client not capable of the API. Handle error. */ }

PublicKeyCredential.isUserVerifyingPlatformAuthenticatorAvailable()
    .then(function (uvpaAvailable) {
        // If there is a user-verifying platform authenticator
        if (uvpaAvailable) {
            // Render some RP-specific UI and get a Promise for a Boolean value
            return askIfUserWantsToCreateCredential();
        }
    }).then(function (userSaidYes) {
        // If there is a user-verifying platform authenticator
        // AND the user wants to create a credential
        if (userSaidYes) {
            var publicKeyOptions = { /* Public key credential creation options. */};
            return navigator.credentials.create({ "publicKey": publicKeyOptions });
        }
    }).then(function (newCredentialInfo) {
        if (newCredentialInfo) {
            // Send new credential info to server for verification and registration.
        }
    }).catch(function (err) {
        // Something went wrong. Handle appropriately.
    });
```

§ 1.3.3. 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 for an Authentication Assertion, providing as much information as possible to narrow the choice of acceptable credentials for the user. This can 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 connects to the authenticator, performing any pairing actions if necessary.
6. The authenticator presents the user with a notification that their attention is needed. 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, 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 in its database, and verifies the assertion signature. If valid, it looks up the identity associated with the assertion's credential ID; that identity is now authenticated. If the credential ID is not recognized by the server (e.g., it has been deregistered due to inactivity) then the authentication has failed; each Relying Party will handle this in its own way.

- The server now does whatever it would otherwise do upon successful authentication -- return a success page, set authentication cookies, etc.

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

```
EXAMPLE 3
if (!window.PublicKeyCredential) { /* Client not capable. Handle error. */ }

// credentialId is generated by the authenticator and is an opaque random byte array
var credentialId = new Uint8Array([183, 148, 245 /* more random bytes previously generated
var options = {
  // The challenge is produced by the server; see the Security Considerations
  challenge: new Uint8Array([4,101,15 /* 29 more random bytes generated by the server */])
  timeout: 120000,  // 2 minutes
  allowCredentials: [{ type: "public-key", id: credentialId }]
};

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

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

Credential Properties Extension.

```
EXAMPLE 4
if (!window.PublicKeyCredential) { /* Client not capable. Handle error. */ }

var encoder = new TextEncoder();
var acceptableCredential1 = {
    type: "public-key",
    id: encoder.encode("BA44712732CE")
};
var acceptableCredential2 = {
    type: "public-key",
    id: encoder.encode("BG35122345NF")
};

var options = {
  // The challenge is produced by the server; see the Security Considerations
  challenge: new Uint8Array([8,18,33 /* 29 more random bytes generated by the server */]),
  timeout: 120000,  // 2 minutes
  allowCredentials: [acceptableCredential1, acceptableCredential2],
  extensions: { 'credProps': true }
};

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

§ 1.3.4. Aborting Authentication Operations

The below example shows how a developer may use the AbortSignal parameter to abort a credential registration operation.

A similar procedure applies to an authentication operation.

```
EXAMPLE 5
const authAbortController = new AbortController();
const authAbortSignal = authAbortController.signal;

authAbortSignal.onabort = function () {
    // Once the page knows the abort started, inform user it is attempting to abort.
}

var options = {
    // A list of options.
}

navigator.credentials.create({
    publicKey: options,
    signal: authAbortSignal})
    .then(function (attestation) {
        // Register the user.
    }).catch(function (error) {
        if (error == "AbortError") {
            // Inform user the credential hasn't been created.
            // Let the server know a key hasn't been created.
        }
    });

// Assume widget shows up whenever authentication occurs.
if (widget == "disappear") {
    authAbortController.abort();
}
```

§ 1.3.5. Decommissioning

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

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

§ 1.4. Platform-Specific Implementation Guidance

This specification defines how to use Web Authentication in the general case. When using Web Authentication in connection with specific platform support (e.g. apps), it is recommended to see platform-specific documentation and guides for additional guidance and limitations.

§ 2. Conformance

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

§ 2.1. User Agents

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

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

§ 2.1.1. Enumerations as DOMString types

Enumeration types are not referenced by other parts of the Web IDL because that would preclude other values from being used without updating this specification and its implementations. It is important for backwards compatibility that client platforms and Relying Parties handle unknown values. Enumerations for this specification exist here for documentation and

as a registry. Where the enumerations are represented elsewhere, they are typed as `DOMString`s, for example in `transports`.

§ 2.2. Authenticators

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

As described in § 1.2 Use Cases, an authenticator may be implemented in the operating system underlying the User Agent, or in external hardware, or a combination of both.

§ 2.2.1. Backwards Compatibility with FIDO U2F

Authenticators that only support the § 8.6 FIDO U2F Attestation Statement Format have no mechanism to store a user handle, so the returned `userHandle` will always be null.

§ 2.3. WebAuthn Relying Parties

A WebAuthn Relying Party MUST behave as described in § 7 WebAuthn Relying Party Operations to obtain all the security benefits offered by this specification. See § 13.4.1 Security Benefits for WebAuthn Relying Parties for further discussion of this.

§ 2.4. All Conformance Classes

All CBOR encoding performed by the members of the above conformance classes MUST be done using the CTAP2 canonical CBOR encoding form. All decoders of the above conformance classes SHOULD reject CBOR that is not validly encoded in the CTAP2 canonical CBOR encoding form and SHOULD reject messages with duplicate map keys.

§ 3. Dependencies

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

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

CBOR
: A number of structures in this specification, including attestation statements and extensions, are encoded using the CTAP2 canonical CBOR encoding form of the Compact Binary Object Representation (***CBOR***) [RFC8949], as defined in [FIDO-CTAP].

CDDL
: This specification describes the syntax of all CBOR-encoded data using the CBOR Data Definition Language (***CDDL***) [RFC8610].

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

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

DOM

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

ECMAScript

%ArrayBuffer% is defined in [ECMAScript].

HTML

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

URL

The concept of same site is defined in [URL].

Web IDL

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

FIDO AppID

The algorithms for determining the FacetID of a calling application and determining if a caller's FacetID is authorized for an AppID (used only in the AppID extension) are defined by [FIDO-APPID].

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

§ 4. Terminology

Attestation

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

during registration. See also § 6.5 Attestation and Figure 6. Whether or how the client conveys the attestation statement and AAGUID portions of the attestation object to the Relying Party is described by attestation conveyance.

Attestation Certificate

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

Authentication

Authentication Ceremony

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

The WebAuthn authentication ceremony is defined in § 7.2 Verifying an Authentication Assertion, and is initiated by the Relying Party calling `navigator.credentials.get()` with a `publicKey` argument. See § 5 Web Authentication API for an introductory overview and § 1.3.3 Authentication for implementation examples.

Authentication Assertion

Assertion

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

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

Authenticator

WebAuthn Authenticator

A cryptographic entity, existing in hardware or software, that can register a user with a given Relying Party and later assert possession of the registered public key credential, and optionally verify the user, when requested by the Relying

Party. Authenticators can report information regarding their type and security characteristics via attestation during registration.

A WebAuthn Authenticator could be a roaming authenticator, a dedicated hardware subsystem integrated into the client device, or a software component of the client or client device.

In general, an authenticator is assumed to have only one user. If multiple natural persons share access to an authenticator, they are considered to represent the same user in the context of that authenticator. If an authenticator implementation supports multiple users in separated compartments, then each compartment is considered a separate authenticator with a single user with no access to other users' credentials.

Authorization Gesture

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

Biometric Recognition

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

Biometric Authenticator

Any authenticator that implements biometric recognition.

Bound credential

A public key credential source or public key credential is said to be bound to its managing authenticator. This means that only the managing authenticator can generate assertions for the public key credential sources bound to it.

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
WebAuthn Client

Also referred to herein as simply a client. See also Conforming User Agent. A WebAuthn Client is an intermediary entity typically implemented in the user agent (in whole, or in part). Conceptually, it underlies the Web Authentication API and embodies the implementation of the `[[Create]](origin, options, sameOriginWithAncestors)` and `[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)` internal methods. It is responsible for both marshalling the inputs for the underlying authenticator operations, and for returning the results of the latter operations to the Web Authentication API's callers.

The WebAuthn Client runs on, and is distinct from, a WebAuthn Client Device.

Client Device
WebAuthn Client Device

The hardware device on which the WebAuthn Client runs, for example a smartphone, a laptop computer or a desktop computer, and the operating system running on that hardware.

The distinctions between a WebAuthn Client device and a client are:

- a single client device MAY support running multiple clients, i.e., browser implementations, which all have access to the same authenticators available on that client device, and
- platform authenticators are bound to a client device rather than a WebAuthn Client.

A client device and a client together constitute a client platform.

Client Platform

A client device and a client together make up a client platform. A single hardware device MAY be part of multiple distinct client platforms at different times by running different operating systems and/or clients.

Client-Side

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

Client-side discoverable Public Key Credential Source
Client-side discoverable Credential
Discoverable Credential
[DEPRECATED] ***Resident Credential***
[DEPRECATED] ***Resident Key***

> Note: Historically, client-side discoverable credentials have been known as resident credentials or resident keys. Due to the phrases `ResidentKey` and `residentKey` being widely used in both the WebAuthn API and also in the Authenticator Model (e.g., in dictionary member names, algorithm variable names, and operation parameters) the usage of `resident` within their names has not been changed for backwards compatibility purposes. Also, the term resident key is defined here as equivalent to a client-side discoverable credential.

A Client-side discoverable Public Key Credential Source, or Discoverable Credential for short, is a public key credential source that is ***discoverable*** and usable in authentication ceremonies where the Relying Party does not provide any credential IDs, i.e., the Relying Party invokes `navigator.credentials.get()` with an ***empty*** `allowCredentials` argument. This means that the Relying Party does not necessarily need to first identify the user.

As a consequence, a discoverable credential capable authenticator can generate an assertion signature for a discoverable credential given only an RP ID, which in turn necessitates that the public key credential source is stored in the authenticator or client platform. This is in contrast to a Server-side Public Key Credential Source, which requires that the authenticator is given both the RP ID and the credential ID but does not require client-side storage of the public key credential source.

See also: client-side credential storage modality and non-discoverable credential.

> Note: Client-side discoverable credentials are also usable in authentication ceremonies where credential IDs are given, i.e., when calling `navigator.credentials.get()` with a non-empty `allowCredentials` argument.

Conforming User Agent

A user agent implementing, in cooperation with the underlying client device, the Web Authentication API and algorithms given in this specification, and handling communication between authenticators and Relying Parties.

Credential ID

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

Credential IDs are generated by authenticators in two forms:

1. At least 16 bytes that include at least 100 bits of entropy, or
2. The public key credential source, without its Credential ID or mutable items, encrypted so only its managing authenticator can decrypt it. This form allows the authenticator to be nearly stateless, by having the Relying Party store any necessary state.

   > Note: [FIDO-UAF-AUTHNR-CMDS] includes guidance on encryption techniques under "Security Guidelines".

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

Credential Key Pair
Credential Private Key
Credential Public Key
User Public Key

A credential key pair is a pair of asymmetric cryptographic keys generated by an authenticator and scoped to a specific WebAuthn Relying Party. It is the central part of a public key credential.

A credential public key is the public key portion of a credential key pair. The credential public key is returned to the Relying Party during a registration ceremony.

A credential private key is the private key portion of a credential key pair. The credential private key is bound to a particular authenticator - its managing authenticator - and is expected to never be exposed to any other party, not even to the owner of the authenticator.

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.

> Note: The credential public key is referred to as the user public key in FIDO UAF [UAFProtocol], and in FIDO U2F [FIDO-U2F-Message-Formats] and some parts of this specification that relate to it.

Credential Properties

A credential property is some characteristic property of a public key credential source, such as whether it is a client-side discoverable credential or a server-side credential.

Human Palatability

An identifier that is human-palatable is intended to be rememberable and reproducible by typical human users, in contrast to identifiers that are, for example, randomly generated sequences of bits [EduPersonObjectClassSpec].

Non-Discoverable Credential

This is a credential whose credential ID must be provided in `allowCredentials` when calling `navigator.credentials.get()` because it is not client-side discoverable. See also server-side credentials.

Public Key Credential Source

A credential source ([CREDENTIAL-MANAGEMENT-1]) used by an authenticator to generate authentication assertions. A public key credential source consists of a struct with the following items:

type

whose value is of `PublicKeyCredentialType`, defaulting to `public-key`.

id

A Credential ID.

privateKey

The credential private key.

rpId

The Relying Party Identifier, for the Relying Party this public key credential source is scoped to.

userHandle

The user handle associated when this public key credential source was created. This item is nullable.

otherUI

OPTIONAL other information used by the authenticator to inform its UI. For example, this might include the user's `displayName`. otherUI is a ***mutable item*** and SHOULD NOT be bound to the public key credential source in a way that prevents otherUI from being updated.

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

Public Key Credential

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

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

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

Rate Limiting

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

Registration

Registration Ceremony

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

employing a test of user presence or user verification. After a successful registration ceremony, the user can be authenticated by an authentication ceremony.

The WebAuthn registration ceremony is defined in § 7.1 Registering a New Credential, and is initiated by the Relying Party calling `navigator.credentials.create()` with a `publicKey` argument. See § 5 Web Authentication API for an introductory overview and § 1.3.1 Registration for implementation examples.

Relying Party

See WebAuthn Relying Party.

Relying Party Identifier
RP ID

In the context of the WebAuthn API, a relying party identifier is a valid domain string identifying the WebAuthn Relying Party on whose behalf a given registration or authentication ceremony is being performed. A public key credential can only be used for authentication with the same entity (as identified by RP ID) it was registered with.

By default, the RP ID for a WebAuthn operation is set to the caller's origin's effective domain. This default MAY be overridden by the caller, as long as the caller-specified RP ID value is a registrable domain suffix of or is equal to the caller's origin's effective domain. See also § 5.1.3 Create a New Credential - PublicKeyCredential's [[Create]](origin, options, sameOriginWithAncestors) Method and § 5.1.4 Use an Existing Credential to Make an Assertion - PublicKeyCredential's [[Get]](options) Method.

Note: An RP ID is based on a host's domain name. It does not itself include a scheme or port, as an origin does. The RP ID of a public key credential determines its ***scope***. I.e., it ***determines the set of origins on which the public key credential may be exercised***, as follows:

- The RP ID must be equal to the origin's effective domain, or a registrable domain suffix of the origin's effective domain.
- The origin's scheme must be `https`.
- The origin's port is unrestricted.

For example, given a Relying Party whose origin is `https://login.example.com:1337`, then the following RP IDs are valid: `login.example.com` (default) and `example.com`, but not `m.login.example.com` and not `com`.

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.

These restrictions on origin values apply to WebAuthn Clients.

Other specifications mimicking the WebAuthn API to enable WebAuthn public key credentials on non-Web platforms (e.g. native mobile applications), MAY define different rules for binding a caller to a Relying Party Identifier. Though, the RP ID syntaxes MUST conform to either valid domain strings or URIs [RFC3986] [URL].

Server-side Public Key Credential Source
Server-side Credential
[DEPRECATED] ***Non-Resident Credential***

Note: Historically, server-side credentials have been known as non-resident credentials. For backwards compatibility purposes, the various WebAuthn API and Authenticator Model components with various forms of `resident` within their names have not been changed.

A Server-side Public Key Credential Source, or Server-side Credential for short, is a public key credential source that is only usable in an authentication ceremony when the Relying Party supplies its credential ID in `navigator.credentials.get()`'s `allowCredentials` argument. This means that the Relying Party must manage the credential's storage and discovery, as well as be able to first identify the user in order to discover the credential IDs to supply in the `navigator.credentials.get()` call.

Client-side storage of the public key credential source is not required for a server-side credential. This is in contrast to a client-side discoverable credential, which instead does not require the user to first be identified in order to provide the user's credential IDs to a `navigator.credentials.get()` call.

See also: server-side credential storage modality and non-discoverable credential.

Test of User Presence

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

User Consent

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

User Handle

The user handle is specified by a Relying Party, as the value of `user.id`, and used to map a specific public key credential to a specific user account with the Relying Party. Authenticators in turn map RP IDs and user handle pairs to public key credential sources.

A user handle is an opaque byte sequence with a maximum size of 64 bytes, and is not meant to be displayed to the user.

User Verification

The technical process by which an authenticator *locally authorizes* the invocation of the authenticatorMakeCredential and authenticatorGetAssertion operations. User verification MAY be instigated through various authorization gesture modalities; for example, through a touch plus pin code, password entry, or biometric recognition (e.g., presenting a fingerprint) [ISOBiometricVocabulary]. The intent is to distinguish individual users.

Note that user verification does not give the Relying Party a concrete identification of the user, but when 2 or more ceremonies with user verification have been done with that credential it expresses that it was the same user that performed all of them. The same user might not always be the same natural person, however, if multiple natural persons share access to the same authenticator.

> Note: Distinguishing natural persons depends in significant part upon the client platform's and authenticator's capabilities. For example, some devices are intended to be used by a single individual, yet they may allow multiple natural persons to enroll fingerprints or know the same PIN and thus access the same Relying Party account(s) using that device.

> Note: Invocation of the authenticatorMakeCredential and authenticatorGetAssertion operations implies use of key material managed by the authenticator.
>
> Also, for security, user verification and use of credential private keys must all occur within the logical security boundary defining the authenticator.

User verification procedures MAY implement rate limiting as a protection against brute force attacks.

User Present
UP

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

User Verified
UV

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

WebAuthn Relying Party

The entity whose ***web application*** utilizes the Web Authentication API to register and authenticate users.

A Relying Party implementation typically consists of both some client-side script that invokes the Web Authentication API in the client, and a server-side component that executes the Relying Party operations and other application logic. Communication between the two components MUST use HTTPS or equivalent transport security, but is otherwise beyond the scope of this specification.

> Note: While the term Relying Party is also often 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. In this specification, the term WebAuthn Relying Party is often shortened to be just Relying Party, and explicitly refers to a Relying Party in the WebAuthn context. Note that in any concrete instantiation a WebAuthn context may be embedded in a broader overall context, e.g., one based on OAuth.

§ 5. *Web Authentication API*

This section normatively specifies the API for creating and using public key credentials. The basic idea is that the credentials belong to the user and are managed by a WebAuthn Authenticator, with which the WebAuthn Relying Party interacts through the client platform. Relying Party scripts can (with the user's consent) request the browser to create a new credential for future use by the Relying Party. See Figure 1, below.

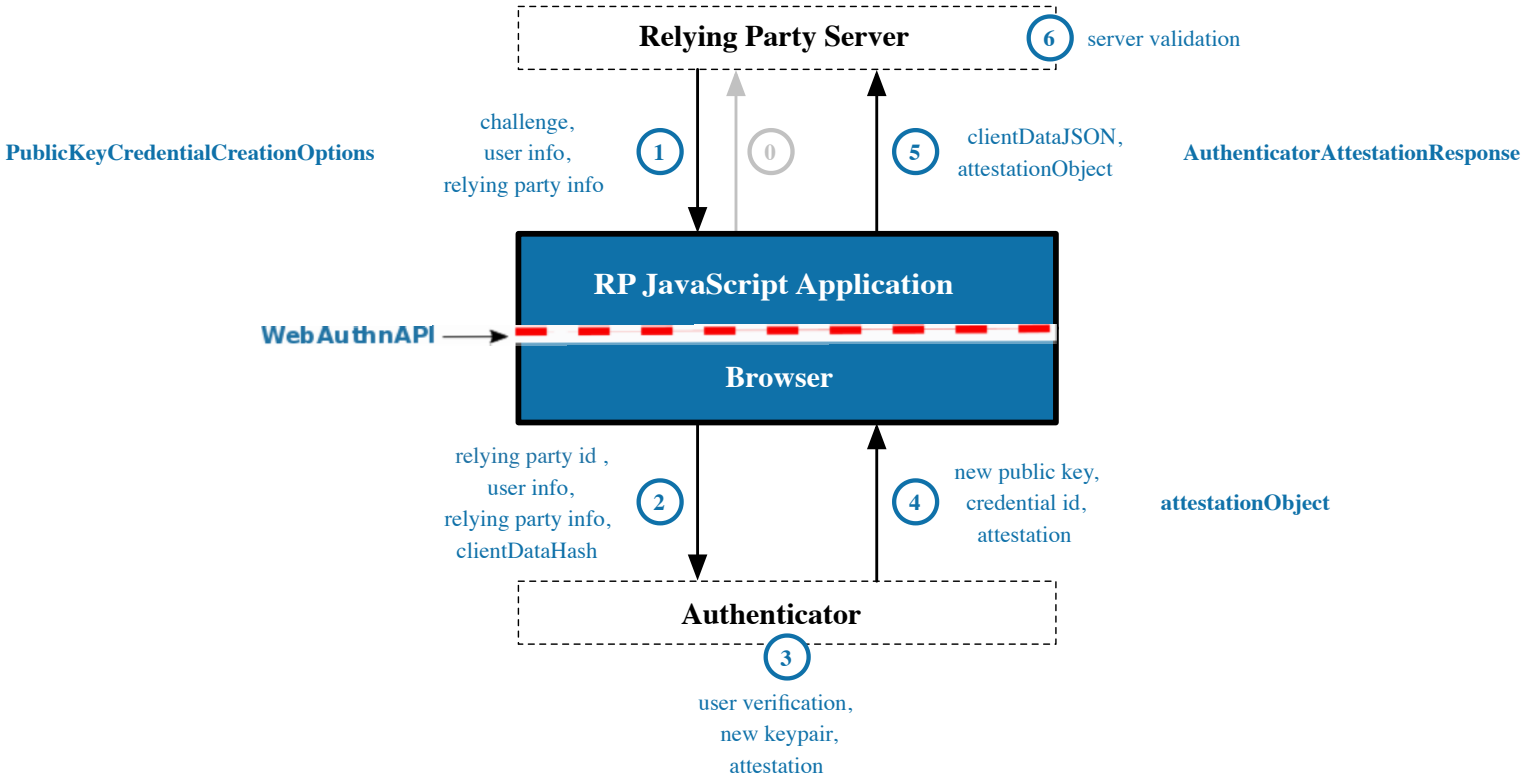

Figure 1 *Registration Flow*

Scripts can also request the user's permission to perform authentication operations with an existing credential. See Figure 2, below.

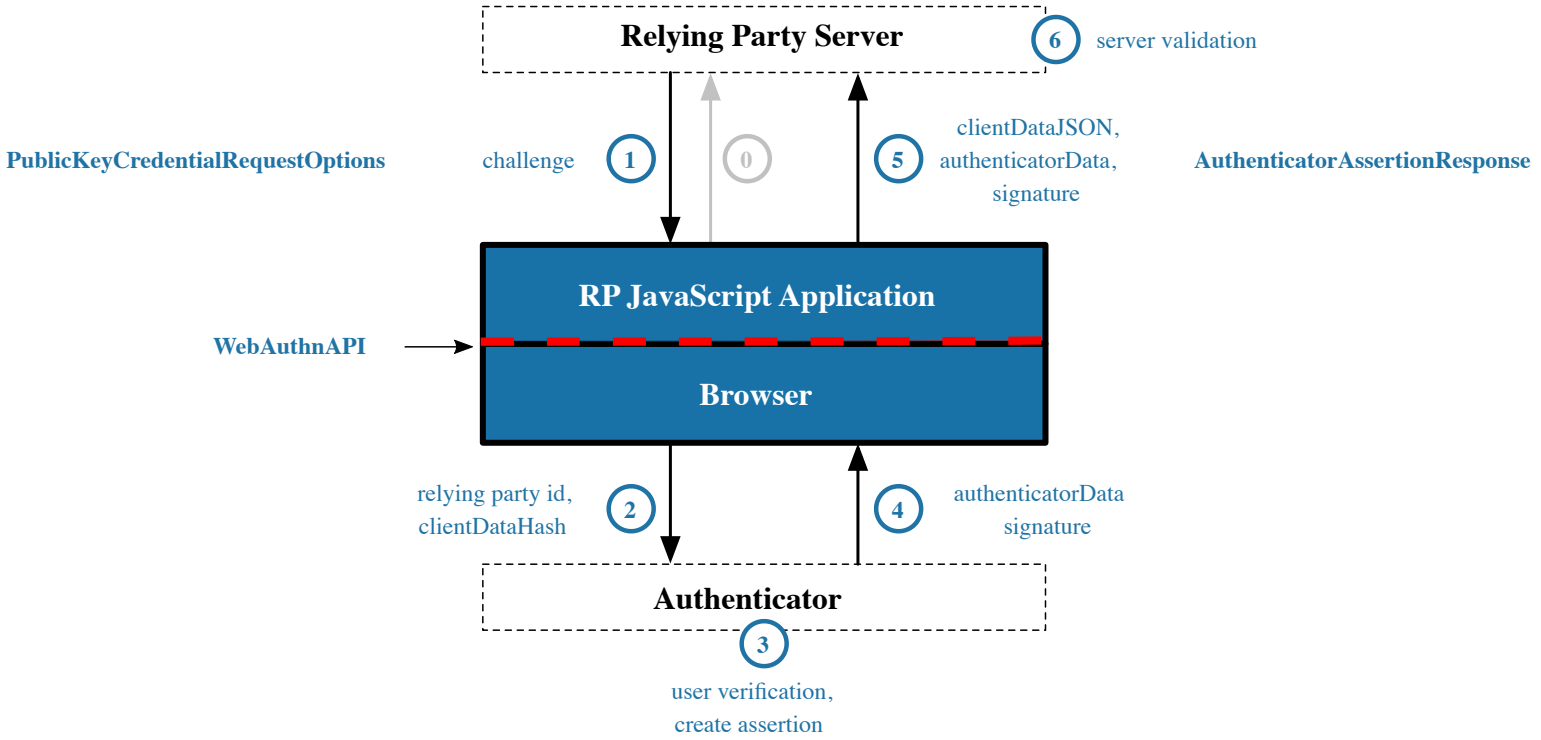

Figure 2 *Authentication Flow*

All such operations are performed in the authenticator and are mediated by the client platform on the user's behalf. At no point does the script get access to the credentials themselves; it only gets information about the credentials in the form of objects.

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

The security properties of this API are provided by the client and the authenticator working together. The authenticator, which holds and manages credentials, ensures that all operations are scoped to a particular origin, and cannot be replayed against a different origin, by incorporating the origin in its responses. Specifically, as defined in § 6.3 Authenticator Operations, the full origin of the requester is included, and signed over, in the attestation object produced when a new credential is created as well as in all assertions produced by WebAuthn credentials.

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

The client facilitates these security measures by providing the Relying Party's origin and RP ID to the authenticator for each operation. Since this is an integral part of the WebAuthn security model, user agents only expose this API to callers in secure contexts. For web contexts in particular, this only includes those accessed via a secure transport (e.g., TLS) established without errors.

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

§ 5.1. `PublicKeyCredential` Interface

✔ MDN

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

✔ MDN

```
[SecureContext, Exposed=Window]
interface PublicKeyCredential : Credential {
    [SameObject] readonly attribute ArrayBuffer             rawId;
```

```
    [SameObject] readonly attribute AuthenticatorResponse    response;
    AuthenticationExtensionsClientOutputs getClientExtensionResults();
};
```

id

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

rawId

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

response, of type AuthenticatorResponse, readonly

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

getClientExtensionResults()

This operation returns the value of `[[clientExtensionsResults]]`, which is a map containing extension identifier → client extension output entries produced by the extension's client extension processing.

[[type]]

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

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

[[discovery]]

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

[[identifier]]

This internal slot contains the credential ID, chosen by the authenticator. The credential ID 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.

> Note: This API does not constrain the format or length of this identifier, except that it MUST be sufficient for the authenticator to uniquely select a key. For example, an authenticator without on-board storage may create identifiers containing a credential private key wrapped with a symmetric key that is burned into the authenticator.

[[clientExtensionsResults]]

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

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

5.1.1. `CredentialCreationOptions` Dictionary Extension

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

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

§ 5.1.2. `CredentialRequestOptions` Dictionary Extension

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

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

§ 5.1.3. Create a New Credential - PublicKeyCredential's `[[Create]](origin, options, sameOriginWithAncestors)` Method

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

This internal method accepts three arguments:

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

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

sameOriginWithAncestors

This argument is a Boolean value which is `true` if and only if the caller's environment settings object is same-origin with its ancestors. It is `false` if caller is cross-origin.

> Note: Invocation of this internal method indicates that it was allowed by permissions policy, which is evaluated at the [CREDENTIAL-MANAGEMENT-1] level. See § 5.9 Permissions Policy integration.

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

> Note: All `BufferSource` objects used in this algorithm must be snapshotted when the algorithm begins, to avoid potential synchronization issues. The algorithm implementations should get a copy of the bytes held by the buffer source and use that copy for relevant portions of the algorithm.

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

1. Assert: *`options`*`.publicKey` is present.
2. If *sameOriginWithAncestors* is `false`, return a "`NotAllowedError`" `DOMException`.

   > Note: This "sameOriginWithAncestors" restriction aims to address a tracking concern raised in Issue #1336. This may be revised in future versions of this specification.

3. Let *options* be the value of *`options`*`.publicKey`.
4. If the `timeout` member of *options* is present, check if its value lies within a reasonable range as defined by the client and if not, correct it to the closest value lying within that range. Set a timer *lifetimeTimer* to this adjusted value. If the `timeout` member of *options* is not present, then set *lifetimeTimer* to a client-specific default.

Recommended ranges and defaults for the `timeout` member of *options* are as follows. If *`options`*`.authenticatorSelection.userVerification`

↪ **is set to `discouraged`**

Recommended range: 30000 milliseconds to 180000 milliseconds.

Recommended default value: 120000 milliseconds (2 minutes).

↪ **is set to `required` or `preferred`**

Recommended range: 30000 milliseconds to 600000 milliseconds.

Recommended default value: 300000 milliseconds (5 minutes).

> Note: The user agent should take cognitive guidelines into considerations regarding timeout for users with special needs.

5. If the length of *`options`*`.user.id` is not between 1 and 64 bytes (inclusive) then return a `TypeError`.
6. Let *callerOrigin* be `origin`. If *callerOrigin* is an opaque origin, return a `DOMException` whose name is "`NotAllowedError`", and terminate this algorithm.
7. Let *effectiveDomain* be the *callerOrigin*'s effective domain. If effective domain is not a valid domain, then return a `DOMException` whose name is "`SecurityError`" and terminate this algorithm.

> Note: An effective domain may resolve to a host, which can be represented in various manners, such as domain, ipv4 address, ipv6 address, opaque host, or empty host. Only the domain format of host is allowed here. This is for simplification and also is in recognition of various issues with using direct IP address identification in concert with PKI-based security.

¶ 8. If *`options`*`.rp.id`

↪ **is present**

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

↪ **Is not present**

Set *options*.rp.id to *effectiveDomain*.

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

9. Let *credTypesAndPubKeyAlgs* be a new list whose items are pairs of PublicKeyCredentialType and a COSEAlgorithmIdentifier.
10. If *options*.pubKeyCredParams's size

    ↪ **is zero**

    Append the following pairs of PublicKeyCredentialType and COSEAlgorithmIdentifier values to *credTypesAndPubKeyAlgs*:

    - public-key and -7 ("ES256").
    - public-key and -257 ("RS256").

    ↪ **is non-zero**

    For each *current* of *options*.pubKeyCredParams:

    1. If *current*.type does not contain a PublicKeyCredentialType supported by this implementation, then continue.
    2. Let *alg* be *current*.alg.
    3. Append the pair of *current*.type and *alg* to *credTypesAndPubKeyAlgs*.

If *credTypesAndPubKeyAlgs* is empty, return a `DOMException` whose name is "`NotSupportedError`", and terminate this algorithm.

11. Let *clientExtensions* be a new map and let *authenticatorExtensions* be a new map.
12. If the `extensions` member of *options* is present, then for each *extensionId* → *clientExtensionInput* of *options*.`extensions`:
    1. If *extensionId* is not supported by this client platform or is not a registration extension, then continue.
    2. Set *clientExtensions*[*extensionId*] to *clientExtensionInput*.
    3. If *extensionId* is not an authenticator extension, then continue.
    4. Let *authenticatorExtensionInput* be the (CBOR) result of running *extensionId*'s client extension processing algorithm on *clientExtensionInput*. If the algorithm returned an error, continue.
    5. Set *authenticatorExtensions*[*extensionId*] to the base64url encoding of *authenticatorExtensionInput*.
13. Let *collectedClientData* be a new `CollectedClientData` instance whose fields are:

    `type`
    The string "webauthn.create".

    `challenge`
    The base64url encoding of *options*.`challenge`.

    `origin`
    The serialization of *callerOrigin*.

    `crossOrigin`
    The inverse of the value of the `sameOriginWithAncestors` argument passed to this internal method.

    `tokenBinding`
    The status of Token Binding between the client and the *callerOrigin*, as well as the Token Binding ID associated with *callerOrigin*, if one is available.

14. Let *clientDataJSON* be the JSON-compatible serialization of client data constructed from *collectedClientData*.

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

16. If the `options.signal` is present and its aborted flag is set to `true`, return a `DOMException` whose name is "`AbortError`" and terminate this algorithm.

17. Let *issuedRequests* be a new ordered set.

18. Let *authenticators* represent a value which at any given instant is a set of client platform-specific handles, where each item identifies an authenticator presently available on this client platform at that instant.

    > Note: What qualifies an authenticator as "available" is intentionally unspecified; this is meant to represent how authenticators can be hot-plugged into (e.g., via USB) or discovered (e.g., via NFC or Bluetooth) by the client by various mechanisms, or permanently built into the client.

19. Start *lifetimeTimer*.

20. While *lifetimeTimer* has not expired, perform the following actions depending upon *lifetimeTimer*, and the state and response for each *authenticator* in *authenticators*:

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

    ↪ **If the user exercises a user agent user-interface option to cancel the process,**
    For each *authenticator* in *issuedRequests* invoke the authenticatorCancel operation on *authenticator* and remove *authenticator* from *issuedRequests*. Return a `DOMException` whose name is "`NotAllowedError`".

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

and terminate this algorithm.

↪ **If an *authenticator* becomes available on this client device,**

> Note: This includes the case where an *authenticator* was available upon *lifetimeTimer* initiation.

1. This *authenticator* is now the ***candidate authenticator***.
2. If `options.authenticatorSelection` is present:
    1. If `options.authenticatorSelection.authenticatorAttachment` is present and its value is not equal to *authenticator*'s authenticator attachment modality, continue.
    2. If `options.authenticatorSelection.residentKey`

        ↪ **is present and set to `required`**

        If the *authenticator* is not capable of storing a client-side discoverable public key credential source, continue.

        ↪ **is present and set to `preferred` or `discouraged`**

        No effect.

        ↪ **is not present**

        if `options.authenticatorSelection.requireResidentKey` is set to `true` and the *authenticator* is not capable of storing a client-side discoverable public key credential source, continue.

    3. If `options.authenticatorSelection.userVerification` is set to `required` and the *authenticator* is not capable of performing user verification, continue.
3. Let *requireResidentKey* be the ***effective resident key requirement for credential creation***, a Boolean value, as follows:

    If `options.authenticatorSelection.residentKey`

↪ **is present and set to `required`**

Let *requireResidentKey* be `true`.

↪ **is present and set to `preferred`**

If the *authenticator*

↪ **is capable of client-side credential storage modality**

Let *requireResidentKey* be `true`.

↪ **is not capable of client-side credential storage modality, or if the client cannot determine authenticator capability,**

Let *requireResidentKey* be `false`.

↪ **is present and set to `discouraged`**

Let *requireResidentKey* be `false`.

↪ **is not present**

Let *requireResidentKey* be the value of *`options`*`.authenticatorSelection.requireResidentKey`.

4. Let *userVerification* be the ***effective user verification requirement for credential creation***, a Boolean value, as follows. If *`options`*`.authenticatorSelection.userVerification`

↪ **is set to `required`**

Let *userVerification* be `true`.

↪ **is set to `preferred`**

If the *authenticator*

↪ **is capable of user verification**

Let *userVerification* be `true`.

↪ **is not capable of user verification**

Let *userVerification* be `false`.

↪ **is set to `discouraged`**

Let *userVerification* be `false`.

5. Let *enterpriseAttestationPossible* be a Boolean value, as follows. If `options.attestation`

↪ **is set to `enterprise`**

Let *enterpriseAttestationPossible* be `true` if the user agent wishes to support enterprise attestation for `options.rp.id` (see Step 8, above). Otherwise `false`.

↪ **otherwise**

Let *enterpriseAttestationPossible* be `false`.

6. Let *excludeCredentialDescriptorList* be a new list.

7. For each credential descriptor *C* in `options.excludeCredentials`:

    1. If `C.transports` is not empty, and *authenticator* is connected over a transport not mentioned in `C.transports`, the client MAY continue.

    > Note: If the client chooses to continue, this could result in inadvertently registering multiple credentials bound to the same authenticator if the transport hints in `C.transports` are not accurate. For example, stored transport hints could become inaccurate as a result of software upgrades adding new connectivity options.

    2. Otherwise, Append *C* to *excludeCredentialDescriptorList*.

¶ 3. Invoke the authenticatorMakeCredential operation on *authenticator* with *clientDataHash*, `options.rp`, `options.user`, *requireResidentKey*, *userVerification*, *credTypesAndPubKeyAlgs*, *excludeCredentialDescriptorList*, *enterpriseAttestationPossible*, and *authenticatorExtensions* as parameters.

8. Append *authenticator* to *issuedRequests*.

↪ **If an *authenticator* ceases to be available on this client device,**

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*.

> Note: Authenticators may return an indication of "the user cancelled the entire operation". How a user agent manifests this state to users is unspecified.

↪ **If any *authenticator* returns an error status equivalent to "`InvalidStateError`",**

1. Remove *authenticator* from *issuedRequests*.
2. For each remaining *authenticator* in *issuedRequests* invoke the authenticatorCancel operation on *authenticator* and remove it from *issuedRequests*.
3. Return a `DOMException` whose name is "`InvalidStateError`" and terminate this algorithm.

> Note: This error status is handled separately because the *authenticator* returns it only if *excludeCredentialDescriptorList* identifies a credential bound to the *authenticator* and the user has consented to the operation. Given this explicit consent, it is acceptable for this case to be distinguishable to the Relying Party.

↪ **If any *authenticator* returns an error status not equivalent to "`InvalidStateError`",**

Remove *authenticator* from *issuedRequests*.

> Note: This case does not imply user consent for the operation, so details about the error are hidden from the Relying Party in order to prevent leak of potentially identifying information. See § 14.5.1 Registration Ceremony Privacy for details.

↪ **If any *authenticator* indicates success,**

1. Remove *authenticator* from *issuedRequests*. This authenticator is now the ***selected authenticator***.
2. Let *credentialCreationData* be a struct whose items are:

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

     > Note: this value is `attObj`, as defined in § 6.5.4 Generating an Attestation Object.

   `clientDataJSONResult`
   : whose value is the bytes of *clientDataJSON*.

   `attestationConveyancePreferenceOption`
   : whose value is the value of *options*.`attestation`.

   `clientExtensionResults`
   : whose value is an `AuthenticationExtensionsClientOutputs` object containing extension identifier → client extension output entries. The entries are created by running each extension's client extension processing algorithm to create the client extension outputs, for each client extension in *`options`*`.extensions`.

3. Let *constructCredentialAlg* be an algorithm that takes a global object *global*, and whose steps are:

   1. If *`credentialCreationData`*`.attestationConveyancePreferenceOption`'s value is

      ↪ **"none"**
      : Replace potentially uniquely identifying information with non-identifying versions of the same:

1. If the AAGUID in the attested credential data is 16 zero bytes, *credentialCreationData*.attestationObjectResult.fmt is "packed", and "x5c" is absent from *credentialCreationData*.attestationObjectResult, then self attestation is being used and no further action is needed.

2. Otherwise

    1. Replace the AAGUID in the attested credential data with 16 zero bytes.

    2. Set the value of *credentialCreationData*.attestationObjectResult.fmt to "none", and set the value of *credentialCreationData*.attestationObjectResult.attStmt to be an empty CBOR map. (See § 8.7 None Attestation Statement Format and § 6.5.4 Generating an Attestation Object).

↪ **"indirect"**

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

↪ **"direct" or "enterprise"**

Convey the authenticator's AAGUID and attestation statement, unaltered, to the Relying Party.

2. Let *attestationObject* be a new ArrayBuffer, created using *global*'s %ArrayBuffer%, containing the bytes of *credentialCreationData*.attestationObjectResult's value.

3. Let *id* be *attestationObject*.authData.attestedCredentialData.credentialId.

4. Let *pubKeyCred* be a new PublicKeyCredential object associated with *global* whose fields are:

[[identifier]]

id

response

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

clientDataJSON

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

attestationObject

attestationObject

[[transports]]

A sequence of zero or more unique DOMStrings, in lexicographical order, that the *authenticator* is believed to support. The values SHOULD be members of AuthenticatorTransport, but client platforms MUST ignore unknown values.

If a user agent does not wish to divulge this information it MAY substitute an arbitrary sequence designed to preserve privacy. This sequence MUST still be valid, i.e. lexicographically sorted and free of duplicates. For example, it may use the empty sequence. Either way, in this case the user agent takes the risk that Relying Party behavior may be suboptimal.

If the user agent does not have any transport information, it SHOULD set this field to the empty sequence.

> Note: How user agents discover transports supported by a given authenticator is outside the scope of this specification, but may include information from an attestation certificate (for example [FIDO-Transports-Ext]), metadata communicated in an authenticator protocol such as CTAP2, or special-case knowledge about a platform authenticator.

`[[clientExtensionsResults]]`

A new `ArrayBuffer`, created using *global*'s %ArrayBuffer%, containing the bytes of `credentialCreationData.clientExtensionResults`.

5. Return *pubKeyCred*.

4. For each remaining *authenticator* in *issuedRequests* invoke the authenticatorCancel operation on *authenticator* and remove it from *issuedRequests*.

5. Return *constructCredentialAlg* and terminate this algorithm.

21. Return a `DOMException` whose name is "`NotAllowedError`". In order to prevent information leak that could identify the user without consent, this step MUST NOT be executed before *lifetimeTimer* has expired. See § 14.5.1 Registration Ceremony Privacy for details.

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

§ 5.1.4. Use an Existing Credential to Make an Assertion - PublicKeyCredential's `[[Get]](options)` Method

WebAuthn Relying Parties call `navigator.credentials.get({publicKey:..., ...})` to discover and use an existing public key credential, with the user's consent. Relying Party script optionally specifies some criteria to indicate what credential sources are acceptable to it. The client platform locates credential sources matching the specified criteria, and guides the user to pick one that the script will be allowed to use. The user may choose to decline the entire interaction

even if a credential source is present, for example to maintain privacy. If the user picks a credential source, the user agent then uses § 6.3.3 The authenticatorGetAssertion Operation to sign a Relying Party-provided challenge and other collected data into an assertion, which is used as a credential.

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

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

This `navigator.credentials.get()` operation can be aborted by leveraging the `AbortController`; see DOM §3.3 Using AbortController and AbortSignal objects in APIs for detailed instructions.

§ *5.1.4.1. PublicKeyCredential's `[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)` Method*

This internal method accepts three arguments:

origin

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

options

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

sameOriginWithAncestors

This argument is a Boolean value which is `true` if and only if the caller's environment settings object is same-origin with its ancestors. It is `false` if caller is cross-origin.

> Note: Invocation of this internal method indicates that it was allowed by permissions policy, which is evaluated at the [CREDENTIAL-MANAGEMENT-1] level. See § 5.9 Permissions Policy integration.

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

> Note: All `BufferSource` objects used in this algorithm must be snapshotted when the algorithm begins, to avoid potential synchronization issues. The algorithm implementations should get a copy of the bytes held by the buffer source and use that copy for relevant portions of the algorithm.

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

1. Assert: *`options`*`.publicKey` is present.
2. Let *options* be the value of *`options`*`.publicKey`.
3. If the `timeout` member of *options* is present, check if its value lies within a reasonable range as defined by the client and if not, correct it to the closest value lying within that range. Set a timer *lifetimeTimer* to this adjusted value. If the `timeout` member of *options* is not present, then set *lifetimeTimer* to a client-specific default.

   Recommended ranges and defaults for the `timeout` member of *options* are as follows. If *`options`*`.userVerification`

   ↪ **is set to `discouraged`**

   Recommended range: 30000 milliseconds to 180000 milliseconds.

   Recommended default value: 120000 milliseconds (2 minutes).

↳ **is set to `required` or `preferred`**

Recommended range: 30000 milliseconds to 600000 milliseconds.

Recommended default value: 300000 milliseconds (5 minutes).

> Note: The user agent should take cognitive guidelines into considerations regarding timeout for users with special needs.

4. Let *callerOrigin* be `origin`. If *callerOrigin* is an opaque origin, return a `DOMException` whose name is "`NotAllowedError`", and terminate this algorithm.

5. Let *effectiveDomain* be the *callerOrigin*'s effective domain. If effective domain is not a valid domain, then return a `DOMException` whose name is "`SecurityError`" and terminate this algorithm.

> Note: An effective domain may resolve to a host, which can be represented in various manners, such as domain, ipv4 address, ipv6 address, opaque host, or empty host. Only the domain format of host is allowed here. This is for simplification and also is in recognition of various issues with using direct IP address identification in concert with PKI-based security.

¶ 6. If *options*.`rpId` is not present, then set *rpId* to *effectiveDomain*.

Otherwise:

1. If *options*.`rpId` is not a registrable domain suffix of and is not equal to *effectiveDomain*, return a `DOMException` whose name is "`SecurityError`", and terminate this algorithm.
2. Set *rpId* to *options*.`rpId`.

> Note: *rpId* represents the caller's RP ID. The RP ID defaults to being the caller's origin's effective domain unless the caller has explicitly set *options*.`rpId` when calling `get()`.

7. Let *clientExtensions* be a new map and let *authenticatorExtensions* be a new map.

8. If the `extensions` member of *options* is present, then for each *extensionId* → *clientExtensionInput* of *options*.`extensions`:

    1. If *extensionId* is not supported by this client platform or is not an authentication extension, then continue.
    2. Set *clientExtensions*[*extensionId*] to *clientExtensionInput*.
    3. If *extensionId* is not an authenticator extension, then continue.
    4. Let *authenticatorExtensionInput* be the (CBOR) result of running *extensionId*'s client extension processing algorithm on *clientExtensionInput*. If the algorithm returned an error, continue.
    5. Set *authenticatorExtensions*[*extensionId*] to the base64url encoding of *authenticatorExtensionInput*.

9. Let *collectedClientData* be a new `CollectedClientData` instance whose fields are:

    `type`
    : The string "webauthn.get".

    `challenge`
    : The base64url encoding of *options*.`challenge`

    `origin`
    : The serialization of *callerOrigin*.

    `crossOrigin`
    : The inverse of the value of the `sameOriginWithAncestors` argument passed to this internal method.

    `tokenBinding`
    : The status of Token Binding between the client and the *callerOrigin*, as well as the Token Binding ID associated with *callerOrigin*, if one is available.

10. Let *clientDataJSON* be the JSON-compatible serialization of client data constructed from *collectedClientData*.
11. Let *clientDataHash* be the hash of the serialized client data represented by *clientDataJSON*.

12. If the *`options`*`.signal` is present and its aborted flag is set to `true`, return a `DOMException` whose name is "`AbortError`" and terminate this algorithm.

13. Let *issuedRequests* be a new ordered set.

14. Let *savedCredentialIds* be a new map.

15. Let *authenticators* represent a value which at any given instant is a set of client platform-specific handles, where each item identifies an authenticator presently available on this client platform at that instant.

    > Note: What qualifies an authenticator as "available" is intentionally unspecified; this is meant to represent how authenticators can be hot-plugged into (e.g., via USB) or discovered (e.g., via NFC or Bluetooth) by the client by various mechanisms, or permanently built into the client.

16. Start *lifetimeTimer*.

17. While *lifetimeTimer* has not expired, perform the following actions depending upon *lifetimeTimer*, and the state and response for each *authenticator* in *authenticators*:

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

    ↪ **If the user exercises a user agent user-interface option to cancel the process,**
    For each *authenticator* in *issuedRequests* invoke the authenticatorCancel operation on *authenticator* and remove *authenticator* from *issuedRequests*. Return a `DOMException` whose name is "`NotAllowedError`".

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

↪ **If *issuedRequests* is empty, *`options`*`.allowCredentials` is not empty, and no *authenticator* will become available for any public key credentials therein,**

Indicate to the user that no eligible credential could be found. When the user acknowledges the dialog, return a `DOMException` whose name is "`NotAllowedError`".

> Note: One way a client platform can determine that no *authenticator* will become available is by examining the `transports` members of the present `PublicKeyCredentialDescriptor` items of *`options`*`.allowCredentials`, if any. For example, if all `PublicKeyCredentialDescriptor` items list only `internal`, but all platform *authenticator*s have been tried, then there is no possibility of satisfying the request. Alternatively, all `PublicKeyCredentialDescriptor` items may list `transports` that the client platform does not support.

↪ **If an *authenticator* becomes available on this client device,**

> Note: This includes the case where an *authenticator* was available upon *lifetimeTimer* initiation.

1. If *`options`*`.userVerification` is set to `required` and the *authenticator* is not capable of performing user verification, continue.
2. Let *userVerification* be the ***effective user verification requirement for assertion***, a Boolean value, as follows. If *`options`*`.userVerification`

   ↪ **is set to `required`**

   Let *userVerification* be `true`.

   ↪ **is set to `preferred`**

   If the *authenticator*

   ↪ **is capable of user verification**

   Let *userVerification* be `true`.

↪ **is not capable of user verification**

Let *userVerification* be `false`.

↪ **is set to `discouraged`**

Let *userVerification* be `false`.

3. If *`options`*`.allowCredentials`

↪ **is not empty**

1. Let *allowCredentialDescriptorList* be a new list.
2. Execute a client 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 empty, continue.
4. Let *distinctTransports* be a new ordered set.
5. If *allowCredentialDescriptorList* has exactly one value, set *`savedCredentialIds`*`[`*`authenticator`*`]` to *`allowCredentialDescriptorList`*`[0].id`'s value (see here in § 6.3.3 The authenticatorGetAssertion Operation for more information).
6. For each credential descriptor *C* in *allowCredentialDescriptorList*, append each value, if any, of *`C`*`.transports` to *distinctTransports*.

   > Note: This will aggregate only distinct values of `transports` (for this authenticator) in *distinctTransports* due to the properties of ordered sets.

7. If *distinctTransports*

↪ **is not empty**

The client selects one *transport* value from *distinctTransports*, possibly incorporating local configuration knowledge of the appropriate transport to use with *authenticator* in making its selection.

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

↪ **is empty**

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

↪ **is empty**

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

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

4. Append *authenticator* to *issuedRequests*.

↪ **If an *authenticator* ceases to be available on this client device,**

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*.

> Note: Authenticators may return an indication of "the user cancelled the entire operation". How a user agent manifests this state to users is unspecified.

↪ **If any *authenticator* returns an error status,**

Remove *authenticator* from *issuedRequests*.

↪ **If any *authenticator* indicates success,**

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

credentialIdResult

If `savedCredentialIds[authenticator]` exists, set the value of credentialIdResult to be the bytes of `savedCredentialIds[authenticator]`. Otherwise, set the value of credentialIdResult to be the bytes of the credential ID returned from the successful authenticatorGetAssertion operation, as defined in § 6.3.3 The authenticatorGetAssertion Operation.

clientDataJSONResult

whose value is the bytes of *clientDataJSON*.

authenticatorDataResult

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

signatureResult

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

userHandleResult

If the authenticator returned a user handle, set the value of userHandleResult to be the bytes of the returned user handle. Otherwise, set the value of userHandleResult to null.

clientExtensionResults

whose value is an `AuthenticationExtensionsClientOutputs` object containing extension identifier → client extension output entries. The entries are created by running each extension's client extension processing algorithm to create the client extension outputs, for each client extension in *options*.`extensions`.

3. Let *constructAssertionAlg* be an algorithm that takes a global object *global*, and whose steps are:

    1. Let *pubKeyCred* be a new `PublicKeyCredential` object associated with *global* whose fields are:

        `[[identifier]]`

        A new `ArrayBuffer`, created using *global*'s %ArrayBuffer%, containing the bytes of *assertionCreationData*.`credentialIdResult`.

        `response`

        A new `AuthenticatorAssertionResponse` object associated with *global* whose fields are:

        `clientDataJSON`

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

        `authenticatorData`

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

        `signature`

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

`userHandle`

If *`assertionCreationData`*`.userHandleResult` is null, set this field to null. Otherwise, set this field to a new `ArrayBuffer`, created using *global*'s %ArrayBuffer%, containing the bytes of *`assertionCreationData`*`.userHandleResult`.

`[[clientExtensionsResults]]`

A new `ArrayBuffer`, created using *global*'s %ArrayBuffer%, containing the bytes of *`assertionCreationData`*`.clientExtensionResults`.

2. Return *pubKeyCred*.

4. For each remaining *authenticator* in *issuedRequests* invoke the authenticatorCancel operation on *authenticator* and remove it from *issuedRequests*.

5. Return *constructAssertionAlg* and terminate this algorithm.

18. Return a `DOMException` whose name is "`NotAllowedError`". In order to prevent information leak that could identify the user without consent, this step MUST NOT be executed before *lifetimeTimer* has expired. See § 14.5.2 Authentication Ceremony Privacy for details.

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

§ 5.1.5. Store an Existing Credential - PublicKeyCredential's `[[Store]](credential, sameOriginWithAncestors)` Method

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

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

This internal method accepts two arguments:

credential
: This argument is a `PublicKeyCredential` object.

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

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

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

§ 5.1.6. Preventing Silent Access to an Existing Credential - PublicKeyCredential's `[[preventSilentAccess]](credential, sameOriginWithAncestors)` Method

Calling the ***`[[preventSilentAccess]](credential, sameOriginWithAncestors)`*** method will have no effect on authenticators that require an authorization gesture, but setting that flag may potentially exclude authenticators that can operate without user intervention.

This internal method accepts no arguments.

§ 5.1.7. Availability of User-Verifying Platform Authenticator - PublicKeyCredential's `isUserVerifyingPlatformAuthenticatorAvailable()` Method

WebAuthn Relying Parties use this method to determine whether they can create a new credential using a user-verifying platform authenticator. Upon invocation, the client employs a client platform-specific procedure to discover available user-verifying platform authenticators. If any are discovered, the promise is resolved with the value of `true`. Otherwise, the promise is resolved with the value of `false`. Based on the result, the Relying Party can take further actions to guide the user to create a credential.

This method has no arguments and returns a Boolean value.

✔ MDN

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

Note: Invoking this method from a browsing context where the Web Authentication API is "disabled" according to the allowed to use algorithm—i.e., by a permissions policy—will result in the promise being rejected with a `DOMException` whose name is "`NotAllowedError`". See also § 5.9 Permissions Policy integration.

§ 5.2. Authenticator Responses (interface ***`AuthenticatorResponse`***)

✔ MDN

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

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

```
};
```

***clientDataJSON*, of type ArrayBuffer, readonly**

✔MDN

This attribute contains a JSON-compatible serialization of the client data, the hash of which is passed to the authenticator by the client in its call to either `create()` or `get()` (i.e., the client data itself is not sent to the authenticator).

§ 5.2.1. Information About Public Key Credential (interface *AuthenticatorAttestationResponse*)

✔MDN

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

⚠MDN

```
[SecureContext, Exposed=Window]
interface AuthenticatorAttestationResponse : AuthenticatorResponse {
    [SameObject] readonly attribute ArrayBuffer      attestationObject;
    sequence<DOMString>                              getTransports();
    ArrayBuffer                                      getAuthenticatorData();
    ArrayBuffer?                                     getPublicKey();
    COSEAlgorithmIdentifier                          getPublicKeyAlgorithm();
};
```

`clientDataJSON`

This attribute, inherited from `AuthenticatorResponse`, contains the JSON-compatible serialization of client data (see § 6.5 Attestation) passed to the authenticator by the client in order to generate this credential. The exact JSON serialization MUST be preserved, as the hash of the serialized client data has been computed over it.

attestationObject, of type ArrayBuffer, readonly

This attribute contains an attestation object, which is opaque to, and cryptographically protected against tampering by, the client. The attestation object contains both authenticator data and an attestation statement. The former contains the AAGUID, a unique credential ID, and the credential public key. The contents of the attestation statement are determined by the attestation statement format used by the authenticator. It also contains any additional information that the Relying Party's server requires to validate the attestation statement, as well as to decode and validate the authenticator data along with the JSON-compatible serialization of client data. For more details, see § 6.5 Attestation, § 6.5.4 Generating an Attestation Object, and Figure 6.

`getTransports()`

This operation returns the value of `[[transports]]`.

`getAuthenticatorData()`

This operation returns the authenticator data contained within `attestationObject`. See § 5.2.1.1 Easily accessing credential data.

`getPublicKey()`

This operation returns the DER SubjectPublicKeyInfo of the new credential, or null if this is not available. See § 5.2.1.1 Easily accessing credential data.

`getPublicKeyAlgorithm()`

This operation returns the `COSEAlgorithmIdentifier` of the new credential. See § 5.2.1.1 Easily accessing credential data.

[[transports]]

This internal slot contains a sequence of zero or more unique `DOMString`s in lexicographical order. These values are the transports that the authenticator is believed to support, or an empty sequence if the information is unavailable. The values SHOULD be members of `AuthenticatorTransport` but Relying Parties MUST ignore unknown values.

§ *5.2.1.1. Easily accessing credential data*

Every user of the [[Create]](origin, options, sameOriginWithAncestors) method will need to parse and store the returned credential public key in order to verify future authentication assertions. However, the credential public key is in [RFC8152] (COSE) format, inside the credentialPublicKey member of the attestedCredentialData, inside the authenticator data, inside the attestation object conveyed by AuthenticatorAttestationResponse.attestationObject. Relying Parties wishing to use attestation are obliged to do the work of parsing the attestationObject and obtaining the credential public key because that public key copy is the one the authenticator signed. However, many valid WebAuthn use cases do not require attestation. For those uses, user agents can do the work of parsing, expose the authenticator data directly, and translate the credential public key into a more convenient format.

The getPublicKey() operation thus returns the credential public key as a SubjectPublicKeyInfo. This ArrayBuffer can, for example, be passed to Java's `java.security.spec.X509EncodedKeySpec`, .NET's `System.Security.Cryptography.ECDsa.ImportSubjectPublicKeyInfo`, or Go's `crypto/x509.ParsePKIXPublicKey`.

Use of getPublicKey() does impose some limitations: by using pubKeyCredParams, a Relying Party can negotiate with the authenticator to use public key algorithms that the user agent may not understand. However, if the Relying Party does so, the user agent will not be able to translate the resulting credential public key into SubjectPublicKeyInfo format and the return value of getPublicKey() will be null.

User agents MUST be able to return a non-null value for getPublicKey() when the credential public key has a COSEAlgorithmIdentifier value of:

- -7 (ES256), where kty is 2 (with uncompressed points) and crv is 1 (P-256).
- -257 (RS256).
- -8 (EdDSA), where crv is 6 (Ed25519).

A SubjectPublicKeyInfo does not include information about the signing algorithm (for example, which hash function to use) that is included in the COSE public key. To provide this, `getPublicKeyAlgorithm()` returns the `COSEAlgorithmIdentifier` for the credential public key.

To remove the need to parse CBOR at all in many cases, `getAuthenticatorData()` returns the authenticator data from `attestationObject`. The authenticator data contains other fields that are encoded in a binary format. However, helper functions are not provided to access them because Relying Parties already need to extract those fields when getting an assertion. In contrast to credential creation, where signature verification is optional, Relying Parties should always be verifying signatures from an assertion and thus must extract fields from the signed authenticator data. The same functions used there will also serve during credential creation.

> Note: `getPublicKey()` and `getAuthenticatorData()` were only added in level two of this spec. Relying Parties SHOULD use feature detection before using these functions by testing the value of `'getPublicKey' in AuthenticatorAttestationResponse.prototype`. Relying Parties that require this function to exist may not interoperate with older user-agents.

§ 5.2.2. Web Authentication Assertion (interface *`AuthenticatorAssertionResponse`*)

✔ MDN

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

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

```
    [SameObject] readonly attribute ArrayBuffer      signature;
    [SameObject] readonly attribute ArrayBuffer?     userHandle;
};
```

`clientDataJSON`

This attribute, inherited from `AuthenticatorResponse`, contains the JSON-compatible serialization of client data (see § 5.8.1 Client Data Used in WebAuthn Signatures (dictionary CollectedClientData)) passed to the authenticator by the client in order to generate this assertion. The exact JSON serialization MUST be preserved, as the hash of the serialized client data has been computed over it.

***`authenticatorData`*, of type ArrayBuffer, readonly**

✔ MDN

This attribute contains the authenticator data returned by the authenticator. See § 6.1 Authenticator Data.

***`signature`*, of type ArrayBuffer, readonly**

✔ MDN

This attribute contains the raw signature returned from the authenticator. See § 6.3.3 The authenticatorGetAssertion Operation.

***`userHandle`*, of type ArrayBuffer, readonly, nullable**

✔ MDN

This attribute contains the user handle returned from the authenticator, or null if the authenticator did not return a user handle. See § 6.3.3 The authenticatorGetAssertion Operation.

§ 5.3. Parameters for Credential Generation (dictionary ***`PublicKeyCredentialParameters`***)

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

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

***type*, of type DOMString**

This member specifies the type of credential to be created. The value SHOULD be a member of `PublicKeyCredentialType` but client platforms MUST ignore unknown values, ignoring any `PublicKeyCredentialParameters` with an unknown `type`.

***alg*, of type COSEAlgorithmIdentifier**

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

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

§ 5.4. Options for Credential Creation (dictionary *PublicKeyCredentialCreationOptions*)

✔MDN

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

    required BufferSource                              challenge;
    required sequence<PublicKeyCredentialParameters>   pubKeyCredParams;

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

***rp*, of type PublicKeyCredentialRpEntity**

✔ MDN

This member contains data about the Relying Party responsible for the request.

Its value's `name` member is REQUIRED. See § 5.4.1 Public Key Entity Description (dictionary PublicKeyCredentialEntity) for further details.

Its value's `id` member specifies the RP ID the credential should be scoped to. If omitted, its value will be the `CredentialsContainer` object's relevant settings object's origin's effective domain. See § 5.4.2 Relying Party Parameters for Credential Generation (dictionary PublicKeyCredentialRpEntity) for further details.

***user*, of type PublicKeyCredentialUserEntity**

✔ MDN

This member contains data about the user account for which the Relying Party is requesting attestation.

Its value's `name`, `displayName` and `id` members are REQUIRED. See § 5.4.1 Public Key Entity Description (dictionary PublicKeyCredentialEntity) and § 5.4.3 User Account Parameters for Credential Generation (dictionary PublicKeyCredentialUserEntity) for further details.

***challenge*, of type BufferSource**

✔ MDN

This member contains a challenge intended to be used for generating the newly created credential's attestation object. See the § 13.4.3 Cryptographic Challenges security consideration.

***pubKeyCredParams*, of type sequence<PublicKeyCredentialParameters>**

✔ MDN

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 client makes a best-effort to create the most preferred credential that it can.

***timeout*, of type unsigned long**

✔ MDN

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 client.

***excludeCredentials*, of type sequence<PublicKeyCredentialDescriptor>, defaulting to []** ✔ MDN

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 client 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** ✔ MDN

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

***attestation*, of type DOMString, defaulting to "none"** ✔ MDN

This member is intended for use by Relying Parties that wish to express their preference for attestation conveyance. Its values SHOULD be members of `AttestationConveyancePreference`. Client platforms MUST ignore unknown values, treating an unknown value as if the member does not exist. Its default value is "none".

***extensions*, of type AuthenticationExtensionsClientInputs**

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 § 9 WebAuthn Extensions; consult the IANA "WebAuthn Extension Identifiers" registry [IANA-WebAuthn-Registries] established by [RFC8809] for an up-to-date list of registered WebAuthn Extensions.

§ 5.4.1. Public Key Entity Description (dictionary *PublicKeyCredentialEntity*)

The `PublicKeyCredentialEntity` dictionary describes a user account, or a WebAuthn Relying Party, which a public key credential is associated with or scoped to, respectively.

```
dictionary PublicKeyCredentialEntity {
    required DOMString    name;
};
```

***name*, of type DOMString**

A human-palatable name for the entity. Its function depends on what the `PublicKeyCredentialEntity` represents:

- When inherited by `PublicKeyCredentialRpEntity` it is a human-palatable identifier for the Relying Party, intended only for display. For example, "ACME Corporation", "Wonderful Widgets, Inc." or "ОАО Примертех".
  - Relying Parties SHOULD perform enforcement, as prescribed in Section 2.3 of [RFC8266] for the Nickname Profile of the PRECIS FreeformClass [RFC8264], when setting `name`'s value, or displaying the value to the user.
  - This string MAY contain language and direction metadata. Relying Parties SHOULD consider providing this information. See § 6.4.2 Language and Direction Encoding about how this metadata is encoded.
  - Clients SHOULD perform enforcement, as prescribed in Section 2.3 of [RFC8266] for the Nickname Profile of the PRECIS FreeformClass [RFC8264], on `name`'s value prior to displaying the value to the user or including the value as a parameter of the authenticatorMakeCredential operation.
- When inherited by `PublicKeyCredentialUserEntity`, it is a human-palatable identifier for a user account. It is intended only for display, i.e., aiding the user in determining the difference between user accounts with similar `displayName`s. For example, "alexm", "alex.mueller@example.com" or "+14255551234".
  - The Relying Party MAY let the user choose this value. The Relying Party SHOULD perform enforcement, as prescribed in Section 3.4.3 of [RFC8265] for the UsernameCasePreserved Profile of the PRECIS IdentifierClass [RFC8264], when setting `name`'s value, or displaying the value to the user.
  - This string MAY contain language and direction metadata. Relying Parties SHOULD consider providing this information. See § 6.4.2 Language and Direction Encoding about how this metadata is encoded.
  - Clients SHOULD perform enforcement, as prescribed in Section 3.4.3 of [RFC8265] for the UsernameCasePreserved Profile of the PRECIS IdentifierClass [RFC8264], on `name`'s value prior to

displaying the value to the user or including the value as a parameter of the authenticatorMakeCredential operation.

When clients, client platforms, or authenticators display a `name`'s value, they should always use UI elements to provide a clear boundary around the displayed value, and not allow overflow into other elements [css-overflow-3].

Authenticators MAY truncate a `name` member's value so that it fits within 64 bytes, if the authenticator stores the value. See § 6.4.1 String Truncation about truncation and other considerations.

§ 5.4.2. Relying Party Parameters for Credential Generation (dictionary *PublicKeyCredentialRpEntity*)

The `PublicKeyCredentialRpEntity` dictionary is used to supply additional Relying Party attributes when creating a new credential.

```
dictionary PublicKeyCredentialRpEntity : PublicKeyCredentialEntity {
    DOMString      id;
};
```

***id*, of type DOMString**

A unique identifier for the Relying Party entity, which sets the RP ID.

§ 5.4.3. 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 : PublicKeyCredentialEntity {
    required BufferSource   id;
    required DOMString      displayName;
};
```

***id*, of type BufferSource**

The user handle of the user account entity. A user handle is an opaque byte sequence with a maximum size of 64 bytes, and is not meant to be displayed to the user.

To ensure secure operation, authentication and authorization decisions MUST be made on the basis of this `id` member, not the `displayName` nor `name` members. See Section 6.1 of [RFC8266].

The user handle MUST NOT contain personally identifying information about the user, such as a username or e-mail address; see § 14.6.1 User Handle Contents for details. The user handle MUST NOT be empty, though it MAY be null.

> Note: the user handle *ought not* be a constant value across different accounts, even for non-discoverable credentials, because some authenticators always create discoverable credentials. Thus a constant user handle would prevent a user from using such an authenticator with more than one account at the Relying Party.

***displayName*, of type DOMString**

A human-palatable name for the user account, intended only for display. For example, "Alex Müller" or "田中倫". The Relying Party SHOULD let the user choose this, and SHOULD NOT restrict the choice more than necessary.

- Relying Parties SHOULD perform enforcement, as prescribed in Section 2.3 of [RFC8266] for the Nickname Profile of the PRECIS FreeformClass [RFC8264], when setting `displayName`'s value, or displaying the value to

the user.

- This string MAY contain language and direction metadata. Relying Parties SHOULD consider providing this information. See § 6.4.2 Language and Direction Encoding about how this metadata is encoded.
- Clients SHOULD perform enforcement, as prescribed in Section 2.3 of [RFC8266] for the Nickname Profile of the PRECIS FreeformClass [RFC8264], on `displayName`'s value prior to displaying the value to the user or including the value as a parameter of the authenticatorMakeCredential operation.

When clients, client platforms, or authenticators display a `displayName`'s value, they should always use UI elements to provide a clear boundary around the displayed value, and not allow overflow into other elements [css-overflow-3].

Authenticators MUST accept and store a 64-byte minimum length for a `displayName` member's value. Authenticators MAY truncate a `displayName` member's value so that it fits within 64 bytes. See § 6.4.1 String Truncation about truncation and other considerations.

§ 5.4.4. Authenticator Selection Criteria (dictionary *AuthenticatorSelectionCriteria*)

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

```
dictionary AuthenticatorSelectionCriteria {
    DOMString                authenticatorAttachment;
    DOMString                residentKey;
    boolean                  requireResidentKey = false;
    DOMString                userVerification = "preferred";
};
```

***authenticatorAttachment*, of type DOMString**

If this member is present, eligible authenticators are filtered to only authenticators attached with the specified § 5.4.5 Authenticator Attachment Enumeration (enum AuthenticatorAttachment). The value SHOULD be a member of `AuthenticatorAttachment` but client platforms MUST ignore unknown values, treating an unknown value as if the member does not exist.

***residentKey*, of type DOMString**

Specifies the extent to which the Relying Party desires to create a client-side discoverable credential. For historical reasons the naming retains the deprecated “resident” terminology. The value SHOULD be a member of `ResidentKeyRequirement` but client platforms MUST ignore unknown values, treating an unknown value as if the member does not exist. If no value is given then the effective value is `required` if `requireResidentKey` is `true` or `discouraged` if it is `false` or absent.

See `ResidentKeyRequirement` for the description of `residentKey`'s values and semantics.

***requireResidentKey*, of type boolean, defaulting to `false`**

This member is retained for backwards compatibility with WebAuthn Level 1 and, for historical reasons, its naming retains the deprecated “resident” terminology for discoverable credentials. Relying Parties SHOULD set it to `true` if, and only if, `residentKey` is set to `required`.

***userVerification*, of type DOMString, defaulting to `"preferred"`**

This member describes the Relying Party's requirements regarding user verification for the `create()` operation. Eligible authenticators are filtered to only those capable of satisfying this requirement. The value SHOULD be a member of `UserVerificationRequirement` but client platforms MUST ignore unknown values, treating an unknown value as if the member does not exist.

§ 5.4.5. Authenticator Attachment Enumeration (enum *`AuthenticatorAttachment`*)

This enumeration's values describe authenticators' attachment modalities. Relying Parties use this to express a preferred authenticator attachment modality when calling `navigator.credentials.create()` to create a credential.

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

> Note: The `AuthenticatorAttachment` enumeration is deliberately not referenced, see § 2.1.1 Enumerations as DOMString types.

`platform`

This value indicates platform attachment.

`cross-platform`

This value indicates cross-platform attachment.

> Note: An authenticator attachment modality selection option is available only in the `[[Create]](origin, options, sameOriginWithAncestors)` operation. The Relying Party may use it to, for example, ensure the user has a roaming credential for authenticating on another client device; or to specifically register a platform credential for easier reauthentication using a particular client device. The `[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)` operation has no authenticator attachment modality selection option, so the Relying Party SHOULD accept any of the user's registered credentials. The client and user will then use whichever is available and convenient at the time.

§ 5.4.6. Resident Key Requirement Enumeration (enum *ResidentKeyRequirement*)

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

> Note: The `ResidentKeyRequirement` enumeration is deliberately not referenced, see § 2.1.1 Enumerations as DOMString types.

This enumeration's values describe the Relying Party's requirements for client-side discoverable credentials (formerly known as resident credentials or resident keys):

discouraged

This value indicates the Relying Party prefers creating a server-side credential, but will accept a client-side discoverable credential.

> Note: A Relying Party cannot require that a created credential is a server-side credential and the Credential Properties Extension may not return a value for the `rk` property. Because of this, it may be the case that it does not know if a credential is a server-side credential or not and thus does not know whether creating a second credential with the same user handle will evict the first.

preferred

This value indicates the Relying Party strongly prefers creating a client-side discoverable credential, but will accept a server-side credential. For example, user agents SHOULD guide the user through setting up user verification if needed to create a client-side discoverable credential in this case. This takes precedence over the setting of `userVerification`.

required

This value indicates the Relying Party requires a client-side discoverable credential, and is prepared to receive an error if a client-side discoverable credential cannot be created.

> Note: Relying Parties can seek information on whether or not the authenticator created a client-side discoverable credential by inspecting the Credential Properties Extension's return value in light of the value provided for *options*.`authenticatorSelection`.`residentKey`. This is useful when values of `discouraged` or `preferred` are used for *options*.`authenticatorSelection`.`residentKey`, because in those cases it is possible for an authenticator to create *either* a client-side discoverable credential or a server-side credential.

§ 5.4.7. *Attestation Conveyance* Preference Enumeration (enum *AttestationConveyancePreference*)

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

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

> Note: The `AttestationConveyancePreference` enumeration is deliberately not referenced, see § 2.1.1 Enumerations as DOMString types.

none

This value indicates that the Relying Party is not interested in authenticator attestation. For example, in order to potentially avoid having to obtain user consent to relay identifying information to the Relying Party, or to save a roundtrip to an Attestation CA or Anonymization CA.

This is the default value.

indirect

This value indicates that the Relying Party prefers an attestation conveyance yielding verifiable attestation statements, but allows the client to decide how to obtain such attestation statements. The client MAY replace the authenticator-generated attestation statements with attestation statements generated by an Anonymization CA, in order to protect the user's privacy, or to assist Relying Parties with attestation verification in a heterogeneous ecosystem.

> Note: There is no guarantee that the Relying Party will obtain a verifiable attestation statement in this case. For example, in the case that the authenticator employs self attestation.

direct

This value indicates that the Relying Party wants to receive the attestation statement as generated by the authenticator.

enterprise

This value indicates that the Relying Party wants to receive an attestation statement that may include uniquely identifying information. This is intended for controlled deployments within an enterprise where the organization wishes to tie registrations to specific authenticators. User agents MUST NOT provide such an attestation unless the user agent or authenticator configuration permits it for the requested RP ID.

If permitted, the user agent SHOULD signal to the authenticator (at invocation time) that enterprise attestation is requested, and convey the resulting AAGUID and attestation statement, unaltered, to the Relying Party.

§ 5.5. Options for Assertion Generation (dictionary ***PublicKeyCredentialRequestOptions***)

✔MDN

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

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

challenge, **of type BufferSource**

✔MDN

This member represents a challenge that the selected authenticator signs, along with other data, when producing an authentication assertion. See the § 13.4.3 Cryptographic Challenges security consideration.

timeout, **of type unsigned long**

✔MDN

This OPTIONAL member specifies a time, in milliseconds, that the caller is willing to wait for the call to complete. The value is treated as a hint, and MAY be overridden by the client.

rpId, **of type USVString**

✔MDN

This OPTIONAL member specifies the relying party identifier claimed by the caller. If omitted, its value will be the `CredentialsContainer` object's relevant settings object's origin's effective domain.

allowCredentials, **of type sequence<PublicKeyCredentialDescriptor>, defaulting to `[]`**

✔MDN

This OPTIONAL member contains a list of `PublicKeyCredentialDescriptor` objects 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 list).

***userVerification*, of type DOMString, defaulting to "preferred"** ✔ MDN

This OPTIONAL member describes the Relying Party's requirements regarding user verification for the `get()` operation. The value SHOULD be a member of `UserVerificationRequirement` but client platforms MUST ignore unknown values, treating an unknown value as if the member does not exist. Eligible authenticators are filtered to only those capable of satisfying this requirement.

***extensions*, of type AuthenticationExtensionsClientInputs** ✔ MDN

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

§ 5.6. Abort Operations with `AbortSignal`

Developers are encouraged to leverage the `AbortController` to manage the `[[Create]](origin, options, sameOriginWithAncestors)` and `[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)` operations. See DOM §3.3 Using AbortController and AbortSignal objects in APIs section for detailed instructions.

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

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

ISSUE 1 The WHATWG HTML WG is discussing whether to provide a hook when a browsing context gains or loses focuses. If a hook is provided, the above paragraph will be updated to include the hook. See WHATWG HTML WG Issue #2711 for more details.

§ 5.7. WebAuthn Extensions Inputs and Outputs

The subsections below define the data types used for conveying WebAuthn extension inputs and outputs.

Note: Authenticator extension outputs are conveyed as a part of Authenticator data (see Table 1).

Note: The types defined below — `AuthenticationExtensionsClientInputs` and `AuthenticationExtensionsClientOutputs` — are applicable to both registration extensions and authentication extensions. The "Authentication..." portion of their names should be regarded as meaning "WebAuthentication..."

5.7.1. Authentication Extensions Client Inputs (dictionary `AuthenticationExtensionsClientInputs`)

```
dictionary AuthenticationExtensionsClientInputs {
};
```

This is a dictionary containing the client extension input values for zero or more WebAuthn Extensions.

5.7.2. Authentication Extensions Client Outputs (dictionary `AuthenticationExtensionsClientOutputs`)

```
dictionary AuthenticationExtensionsClientOutputs {
};
```

This is a dictionary containing the client extension output values for zero or more WebAuthn Extensions.

§ 5.7.3. Authentication Extensions Authenticator Inputs (CDDL type `AuthenticationExtensionsAuthenticatorInputs`)

```
AuthenticationExtensionsAuthenticatorInputs = {
  * $$extensionInput .within ( tstr => any )
}
```

The CDDL type `AuthenticationExtensionsAuthenticatorInputs` defines a CBOR map containing the authenticator extension input values for zero or more WebAuthn Extensions. Extensions can add members as described in § 9.3 Extending Request Parameters.

This type is not exposed to the Relying Party, but is used by the client and authenticator.

§ 5.7.4. Authentication Extensions Authenticator Outputs (CDDL type `AuthenticationExtensionsAuthenticatorOutputs`)

```
AuthenticationExtensionsAuthenticatorOutputs = {
  * $$extensionOutput .within ( tstr => any )
}
```

The CDDL type `AuthenticationExtensionsAuthenticatorOutputs` defines a CBOR map containing the authenticator extension output values for zero or more WebAuthn Extensions. Extensions can add members as described in § 9.3 Extending Request Parameters.

§ 5.8. Supporting Data Structures

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

§ 5.8.1. Client Data Used in WebAuthn Signatures (dictionary *CollectedClientData*)

The ***client data*** represents the contextual bindings of both the WebAuthn Relying Party and the client. It is a key-value mapping whose keys are strings. Values can be any type that has a valid encoding in JSON. Its structure is defined by the following Web IDL.

> Note: The `CollectedClientData` may be extended in the future. Therefore it's critical when parsing to be tolerant of unknown keys and of any reordering of the keys. See also § 5.8.1.2 Limited Verification Algorithm.

```
dictionary CollectedClientData {
    required DOMString          type;
    required DOMString          challenge;
    required DOMString          origin;
    boolean                     crossOrigin;
    TokenBinding                tokenBinding;
};

dictionary TokenBinding {
    required DOMString status;
    DOMString id;
};

enum TokenBindingStatus { "present", "supported" };
```

type, **of type DOMString**

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

***challenge*, of type DOMString**

This member contains the base64url encoding of the challenge provided by the Relying Party. See the § 13.4.3 Cryptographic Challenges security consideration.

***origin*, of type DOMString**

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

***crossOrigin*, of type boolean**

This member contains the inverse of the `sameOriginWithAncestors` argument value that was passed into the internal method.

***tokenBinding*, of type TokenBinding**

This OPTIONAL member contains information about the state of the Token Binding protocol [TokenBinding] used when communicating with the Relying Party. Its absence indicates that the client doesn't support token binding.

***status*, of type DOMString**

This member SHOULD be a member of `TokenBindingStatus` but client platforms MUST ignore unknown values, treating an unknown value as if the `tokenBinding` member does not exist. When known, this member is one of the following:

supported

Indicates the client supports token binding, but it was not negotiated when communicating with the Relying Party.

present

Indicates token binding was used when communicating with the Relying Party. In this case, the `id` member MUST be present.

> Note: The `TokenBindingStatus` enumeration is deliberately not referenced, see § 2.1.1 Enumerations as DOMString types.

***id*, of type DOMString**

This member MUST be present if `status` is `present`, and MUST be a base64url encoding of the Token Binding ID that was used when communicating with the Relying Party.

> Note: Obtaining a Token Binding ID is a client platform-specific operation.

The `CollectedClientData` structure is used by the client to compute the following quantities:

JSON-compatible serialization of client data

This is the result of performing the JSON-compatible serialization algorithm on the `CollectedClientData` dictionary.

Hash of the serialized client data

This is the hash (computed using SHA-256) of the JSON-compatible serialization of client data, as constructed by the client.

§ *5.8.1.1. Serialization*

The serialization of the `CollectedClientData` is a subset of the algorithm for JSON-serializing to bytes. I.e. it produces a valid JSON encoding of the `CollectedClientData` but also provides additional structure that may be exploited by verifiers to avoid integrating a full JSON parser. While verifiers are recommended to perform standard JSON parsing, they may use the more limited algorithm below in contexts where a full JSON parser is too large. This verification algorithm requires only base64url encoding, appending of bytestrings (which could be implemented by writing into a fixed template), and three conditional checks (assuming that inputs are known not to need escaping).

The serialization algorithm works by appending successive byte strings to an, initially empty, partial result until the complete result is obtained.

1. Let *result* be an empty byte string.

2. Append 0x7b2274797065223a (`{"type":`) to *result*.
3. Append CCDToString(`type`) to *result*.
4. Append 0x2c226368616c6c656e6765223a (`,"challenge":`) to *result*.
5. Append CCDToString(`challenge`) to *result*.
6. Append 0x2c226f726967696e223a (`,"origin":`) to *result*.
7. Append CCDToString(`origin`) to *result*.
8. Append 0x2c2263726f73734f726967696e223a (`,"crossOrigin":`) to *result*.
9. If `crossOrigin` is not present, or is `false`:
    1. Append 0x66616c7365 (`false`) to *result*.
10. Otherwise:
    1. Append 0x74727565 (`true`) to *result*.
11. Create a temporary copy of the `CollectedClientData` and remove the fields `type`, `challenge`, `origin`, and `crossOrigin` (if present).
12. If no fields remain in the temporary copy then:
    1. Append 0x7d (`}`) to *result*.
13. Otherwise:
    1. Invoke serialize JSON to bytes on the temporary copy to produce a byte string *remainder*.
    2. Append 0x2c (`,`) to *result*.
    3. Remove the leading byte from *remainder*.
    4. Append *remainder* to *result*.

14. The result of the serialization is the value of *result*.

The function ***CCDToString*** is used in the above algorithm and is defined as:

1. Let *encoded* be an empty byte string.
2. Append 0x22 (") to *encoded*.
3. Invoke ToString on the given object to convert to a string.
4. For each code point in the resulting string, if the code point:

   ↪ **is in the set {U+0020, U+0021, U+0023–U+005B, U+005D–U+10FFFF}**
   Append the UTF-8 encoding of that code point to *encoded*.

   ↪ **is U+0022**
   Append 0x5c22 (\") to *encoded*.

   ↪ **is U+005C**
   Append 0x5c5c (\\) to *encoded*.

   ↪ **otherwise**
   Append 0x5c75 (\u) to *encoded*, followed by four, lower-case hex digits that, when interpreted as a base-16 number, represent that code point.

5. Append 0x22 (") to *encoded*.
6. The result of this function is the value of *encoded*.

§ *5.8.1.2. Limited Verification Algorithm*

Verifiers may use the following algorithm to verify an encoded `CollectedClientData` if they cannot support a full JSON parser:

1. The inputs to the algorithm are:
    1. A bytestring, *clientDataJSON*, that contains `clientDataJSON` — the serialized `CollectedClientData` that is to be verified.
    2. A string, *type*, that contains the expected `type`.
    3. A byte string, *challenge*, that contains the challenge byte string that was given in the `PublicKeyCredentialRequestOptions` or `PublicKeyCredentialCreationOptions`.
    4. A string, *origin*, that contains the expected `origin` that issued the request to the user agent.
    5. A boolean, *crossOrigin*, that is true if, and only if, the request should have been performed within a cross-origin `<iframe>`.
2. Let *expected* be an empty byte string.
3. Append 0x7b2274797065223a (`{"type":`) to *expected*.
4. Append CCDToString(*type*) to *expected*.
5. Append 0x2c226368616c6c656e6765223a (`,"challenge":`) to *expected*.
6. Perform base64url encoding on *challenge* to produce a string, *challengeBase64*.
7. Append CCDToString(*challengeBase64*) to *expected*.
8. Append 0x2c226f726967696e223a (`,"origin":`) to *expected*.
9. Append CCDToString(*origin*) to *expected*.
10. Append 0x2c2263726f73734f726967696e223a (`,"crossOrigin":`) to *expected*.
11. If *crossOrigin* is true:
    1. Append 0x74727565 (`true`) to *expected*.
12. Otherwise, i.e. *crossOrigin* is false:

1. Append 0x66616c7365 (`false`) to *expected*.

13. If *expected* is not a prefix of *clientDataJSON* then the verification has failed.
14. If *clientDataJSON* is not at least one byte longer than *expected* then the verification has failed.
15. If the byte of *clientDataJSON* at the offset equal to the length of *expected*:

    ↪ **is 0x7d**
    The verification is successful.

    ↪ **is 0x2c**
    The verification is successful.

    ↪ **otherwise**
    The verification has failed.

§ *5.8.1.3. Future development*

In order to remain compatible with the limited verification algorithm, future versions of this specification must not remove any of the fields `type`, `challenge`, `origin`, or `crossOrigin` from `CollectedClientData`. They also must not change the serialization algorithm to change the order in which those fields are serialized.

If additional fields are added to `CollectedClientData` then verifiers that employ the limited verification algorithm will not be able to consider them until the two algorithms above are updated to include them. Once such an update occurs then the added fields inherit the same limitations as described in the previous paragraph. Such an algorithm update would have to accomodate serializations produced by previous versions. I.e. the verification algorithm would have to handle the fact that a fifth key–value pair may not appear fifth (or at all) if generated by a user agent working from a previous version.

§ 5.8.2. Credential Type Enumeration (enum *PublicKeyCredentialType*)

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

Note: The `PublicKeyCredentialType` enumeration is deliberately not referenced, see § 2.1.1 Enumerations as DOMString types.

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

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

§ 5.8.3. Credential Descriptor (dictionary *PublicKeyCredentialDescriptor*)

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

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

***type*, of type DOMString**

This member contains the type of the public key credential the caller is referring to. The value SHOULD be a member of `PublicKeyCredentialType` but client platforms MUST ignore any `PublicKeyCredentialDescriptor` with an unknown `type`.

***id*, of type BufferSource**

This member contains the credential ID of the public key credential the caller is referring to.

***transports*, of type sequence<DOMString>**

This OPTIONAL member contains a hint as to how the client might communicate with the managing authenticator of the public key credential the caller is referring to. The values SHOULD be members of `AuthenticatorTransport` but client platforms MUST ignore unknown values.

The `getTransports()` operation can provide suitable values for this member. When registering a new credential, the Relying Party SHOULD store the value returned from `getTransports()`. When creating a `PublicKeyCredentialDescriptor` for that credential, the Relying Party SHOULD retrieve that stored value and set it as the value of the `transports` member.

§ 5.8.4. Authenticator Transport Enumeration (enum *AuthenticatorTransport*)

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

> Note: The `AuthenticatorTransport` enumeration is deliberately not referenced, see § 2.1.1 Enumerations as DOMString types.

Authenticators may implement various transports for communicating with clients. This enumeration defines hints as to how clients might communicate with a particular authenticator in order to obtain an assertion for a specific credential. Note that these hints represent the WebAuthn Relying Party's best belief as to how an authenticator may be reached. A Relying Party will typically learn of the supported transports for a public key credential via `getTransports()`.

usb
: Indicates the respective authenticator can be contacted over removable USB.

nfc
: Indicates the respective authenticator can be contacted over Near Field Communication (NFC).

ble
: Indicates the respective authenticator can be contacted over Bluetooth Smart (Bluetooth Low Energy / BLE).

internal
: Indicates the respective authenticator is contacted using a client device-specific transport, i.e., it is a platform authenticator. These authenticators are not removable from the client device.

§ 5.8.5. Cryptographic Algorithm Identifier (typedef `COSEAlgorithmIdentifier`)

```
typedef long COSEAlgorithmIdentifier;
```

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

The COSE algorithms registry leaves degrees of freedom to be specified by other parameters in a COSE key. In order to promote interoperability, this specification makes the following additional guarantees of credential public keys:

1. Keys with algorithm ES256 (-7) MUST specify P-256 (1) as the crv parameter and MUST NOT use the compressed point form.
2. Keys with algorithm ES384 (-35) MUST specify P-384 (2) as the crv parameter and MUST NOT use the compressed point form.
3. Keys with algorithm ES512 (-36) MUST specify P-521 (3) as the crv parameter and MUST NOT use the compressed point form.
4. Keys with algorithm EdDSA (-8) MUST specify Ed25519 (6) as the crv parameter. (These always use a compressed form in COSE.)

> Note: There are many checks neccessary to correctly implement signature verification using these algorithms. One of these is that, when processing uncompressed elliptic-curve points, implementations should check that the point is actually on the curve. This check is highlighted because it's judged to be at particular risk of falling through the gap between a cryptographic library and other code.

§ 5.8.6. User Verification Requirement Enumeration (enum *UserVerificationRequirement*)

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

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

> Note: The `UserVerificationRequirement` enumeration is deliberately not referenced, see § 2.1.1 Enumerations as DOMString types.

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

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

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

§ 5.9. Permissions Policy integration

⚠ MDN

This specification defines one policy-controlled feature identified by the feature-identifier token "***publickey-credentials-get***". Its default allowlist is '`self`'. [Permissions-Policy]

A `Document`'s permissions policy determines whether any content in that document is allowed to successfully invoke the Web Authentication API, i.e., via `navigator.credentials.get({publicKey:..., ...})`. If disabled in any document, no content in the document will be allowed to use the foregoing methods: attempting to do so will return an error.

Note: Algorithms specified in [CREDENTIAL-MANAGEMENT-1] perform the actual permissions policy evaluation. This is because such policy evaluation needs to occur when there is access to the current settings object. The `[[Create]](origin, options, sameOriginWithAncestors)` and `[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)` internal methods do not have such access since they are invoked in parallel (by algorithms specified in [CREDENTIAL-MANAGEMENT-1]).

§ 5.10. Using Web Authentication within `iframe` elements

The Web Authentication API is disabled by default in cross-origin `<iframe>`s. To override this default policy and indicate that a cross-origin `<iframe>` is allowed to invoke the Web Authentication API's `[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)` method, specify the `allow` attribute on the `<iframe>` element and include the `publickey-credentials-get` feature-identifier token in the `allow` attribute's value.

Relying Parties utilizing the WebAuthn API in an embedded context should review § 13.4.2 Visibility Considerations for Embedded Usage regarding UI redressing and its possible mitigations.

§ 6. WebAuthn *Authenticator Model*

The Web Authentication API implies a specific abstract functional model for a WebAuthn Authenticator. This section describes that 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 client platform, MUST be indistinguishable from the behavior specified in § 5 Web Authentication API.

> Note: [FIDO-CTAP] is an example of a concrete instantiation of this model, but it is one in which there are differences in the data it returns and those expected by the WebAuthn API's algorithms. The CTAP2 response messages are CBOR maps constructed using integer keys rather than the string keys defined in this specification for the same objects. The client is expected to perform any needed transformations on such data. The [FIDO-CTAP] specification details the mapping between CTAP2 integer keys and WebAuthn string keys, in section §6.2. Responses.

For authenticators, this model defines the logical operations that they MUST support, and the data formats that they expose to the client and the WebAuthn Relying Party. However, it does not define the details of how authenticators communicate with the client device, unless they are necessary for interoperability with Relying Parties. For instance, this abstract model does not define protocols for connecting authenticators to clients over transports such as USB or NFC. Similarly, this abstract model does not define specific error codes or methods of returning them; however, it does define error behavior in terms of the needs of the client. Therefore, specific error codes are mentioned as a means of showing which error conditions MUST be distinguishable (or not) from each other in order to enable a compliant and secure client implementation.

Relying Parties may influence authenticator selection, if they deem necessary, by stipulating various authenticator characteristics when creating credentials and/or when generating assertions, through use of credential creation options or assertion generation options, respectively. The algorithms underlying the WebAuthn API marshal these options and pass them to the applicable authenticator operations defined below.

In this abstract model, the authenticator provides key management and cryptographic signatures. It can be embedded in the WebAuthn client or housed in a separate device entirely. The authenticator itself can 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 a ***credentials map***, a map from (rpId, [userHandle]) to public key credential source.

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

authenticators made by that manufacturer, and different (with high probability) from the AAGUIDs of all other types of authenticators. The AAGUID for a given type of authenticator SHOULD be randomly generated to ensure this. The Relying Party MAY use the AAGUID to infer certain properties of the authenticator, such as certification level and strength of key protection, using information from other sources.

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

The goals of this design can be summarized as follows.

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

Authenticators produce cryptographic signatures for two distinct purposes:

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

credential public key. The attestation signature is signed by an attestation private key, which is chosen depending on the type of attestation desired. For more details on attestation, see § 6.5 Attestation.

2. An ***assertion signature*** is produced when the authenticatorGetAssertion method is invoked. It represents an assertion by the authenticator that the user has consented to a specific transaction, such as logging in, or completing a purchase. Thus, an assertion signature asserts that the authenticator possessing a particular credential private key has established, to the best of its ability, that the user requesting this transaction is the same user who consented to creating that particular public key credential. It also asserts additional information, termed client data, that may be useful to the caller, such as the means by which user consent was provided, and the prompt shown to the user by the authenticator. The assertion signature format is illustrated in Figure 4, below.

The term ***WebAuthn signature*** refers to both attestation signatures and assertion signatures. The formats of these signatures, as well as the procedures for generating them, are specified below.

§ 6.1. Authenticator Data

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

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

The authenticator data structure is a byte array of 37 bytes or more, laid out as shown in Table 1.

Name	Length (in bytes)	Description
rpIdHash	32	SHA-256 hash of the RP ID the credential is scoped to.

flags	1	Flags (bit 0 is the least significant bit):
		• Bit 0: User Present (UP) result.
		  ◦ `1` means the user is present.
		  ◦ `0` means the user is not present.
		• Bit 1: Reserved for future use (`RFU1`).
		• Bit 2: User Verified (UV) result.
		  ◦ `1` means the user is verified.
		  ◦ `0` means the user is not verified.
		• Bits 3-5: Reserved for future use (`RFU2`).
		• Bit 6: Attested credential data included (`AT`).
		  ◦ Indicates whether the authenticator added attested credential data.
		• Bit 7: Extension data included (`ED`).
		  ◦ Indicates if the authenticator data has extensions.

signCount	4	Signature counter, 32-bit unsigned big-endian integer.

attestedCredentialData	variable (if present)	attested credential data (if present). See § 6.5.1 Attested Credential Data for details. Its length depends on the length of the credential ID and credential public key being attested.
extensions	variable (if present)	Extension-defined authenticator data. This is a CBOR [RFC8949] map with extension identifiers as keys, and authenticator extension outputs as values. See § 9 WebAuthn Extensions for details.

Table 1 *Authenticator data layout. The names in the Name column are only for reference within this document, and are not present in the actual representation of the authenticator data.*

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

Authenticators ***perform the following steps to generate an authenticator data structure***:

- Hash RP ID using SHA-256 to generate the rpIdHash.
- The `UP` flag SHALL be set if and only if the authenticator performed a test of user presence. The `UV` flag SHALL be set if and only if the authenticator performed user verification. The `RFU` bits SHALL be set to zero.

  > Note: If the authenticator performed both a test of user presence and user verification, possibly combined in a single authorization gesture, then the authenticator will set both the `UP` flag and the `UV` flag.

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

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

Figure 3 shows a visual representation of the authenticator data structure.

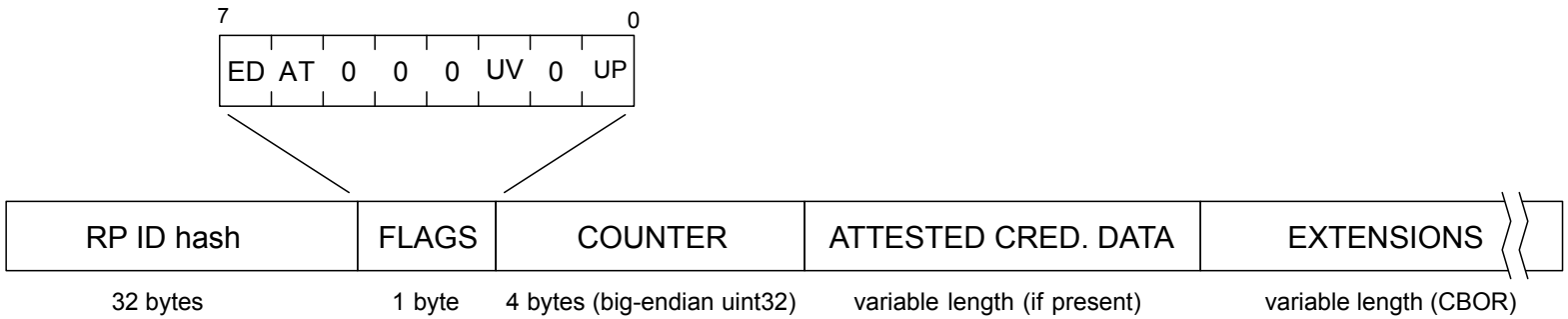

***Figure 3** Authenticator data layout.*

> Note: authenticator data describes its own length: If the AT and ED flags are not set, it is always 37 bytes long. The attested credential data (which is only present if the AT flag is set) describes its own length. If the ED flag is set, then the total length is 37 bytes plus the length of the attested credential data (if the AT flag is set), plus the length of the extensions output (a CBOR map) that follows.
>
> Determining attested credential data's length, which is variable, involves determining `credentialPublicKey`'s beginning location given the preceding `credentialId`'s length, and then determining the `credentialPublicKey`'s length (see also Section 7 of [RFC8152]).

§ 6.1.1. *Signature Counter* Considerations

Authenticators SHOULD implement a signature counter feature. These counters are conceptually stored for each credential by the authenticator, or globally for the authenticator as a whole. The initial value of a credential's signature counter is specified in the `signCount` value of the authenticator data returned by authenticatorMakeCredential. The signature counter is incremented for each successful authenticatorGetAssertion operation by some positive value, and subsequent values are returned to the WebAuthn Relying Party within the authenticator data again. The signature counter's purpose is to aid Relying Parties in detecting cloned authenticators. Clone detection is more important for authenticators with limited protection measures.

A Relying Party stores the signature counter of the most recent authenticatorGetAssertion operation. (Or the counter from the authenticatorMakeCredential operation if no authenticatorGetAssertion has ever been performed on a credential.) In subsequent authenticatorGetAssertion operations, the Relying Party compares the stored signature counter value with the new `signCount` value returned in the assertion's authenticator data. If either is non-zero, and the new `signCount` value is less than or equal to the stored value, a cloned authenticator may exist, or the authenticator may be malfunctioning.

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

Authenticators:

- SHOULD implement per credential signature counters. This prevents the signature counter value from being shared between Relying Parties and being possibly employed as a correlation handle for the user. Authenticators may implement a global signature counter, i.e., on a per-authenticator basis, but this is less privacy-friendly for users.
- SHOULD ensure that the signature counter value does not accidentally decrease (e.g., due to hardware failures).

§ 6.1.2. FIDO U2F Signature Format Compatibility

The format for assertion signatures, which sign over the concatenation of an authenticator data structure and the hash of the serialized client data, are compatible with the FIDO U2F authentication signature format (see Section 5.4 of [FIDO-U2F-Message-Formats]).

This is because the first 37 bytes of the signed data in a FIDO U2F authentication response message constitute a valid authenticator data structure, and the remaining 32 bytes are the hash of the serialized client data. In this authenticator data structure, the `rpIdHash` is the FIDO U2F application parameter, all `flags` except `UP` are always zero, and the `attestedCredentialData` and `extensions` are never present. FIDO U2F authentication signatures can therefore be verified by the same procedure as other assertion signatures generated by the authenticatorMakeCredential operation.

§ 6.2. Authenticator Taxonomy

Many use cases are dependent on the capabilities of the authenticator used. This section defines some terminology for those capabilities, their most important combinations, and which use cases those combinations enable.

For example:

- When authenticating for the first time on a particular client device, a roaming authenticator is typically needed since the user doesn't yet have a platform credential on that client device.
- For subsequent re-authentication on the same client device, a platform authenticator is likely the most convenient since it's built directly into the client device rather than being a separate device that the user may have to locate.
- For second-factor authentication in addition to a traditional username and password, any authenticator can be used.
- Passwordless multi-factor authentication requires an authenticator capable of user verification, and in some cases also discoverable credential capable.

- A laptop computer might support connecting to roaming authenticators via USB and Bluetooth, while a mobile phone might only support NFC.

The above examples illustrate the the primary ***authenticator type*** characteristics:

- Whether the authenticator is a roaming or platform authenticator — the authenticator attachment modality. A roaming authenticator can support one or more transports for communicating with the client.
- Whether the authenticator is capable of user verification — the authentication factor capability.
- Whether the authenticator is discoverable credential capable — the credential storage modality.

These characteristics are independent and may in theory be combined in any way, but Table 2 lists and names some authenticator types of particular interest.

Authenticator Type	Authenticator Attachment Modality	Credential Storage Modality	Authentication Factor Capability
Second-factor platform authenticator	platform	Either	Single-factor capable
User-verifying platform authenticator	platform	Either	Multi-factor capable
Second-factor roaming authenticator	cross-platform	Server-side storage	Single-factor capable
First-factor roaming authenticator	cross-platform	Client-side storage	Multi-factor capable

Table 2 *Definitions of names for some authenticator types.*

A second-factor platform authenticator is convenient to use for re-authentication on the same client device, and can be used to add an extra layer of security both when initiating a new session and when resuming an existing session. A second-factor roaming authenticator is more likely to be used to authenticate on a particular client device for the first time, or on a client device shared between multiple users.

User-verifying platform authenticators and first-factor roaming authenticators enable passwordless multi-factor authentication. In addition to the proof of possession of the credential private key, these authenticators support user verification as a second authentication factor, typically a PIN or biometric recognition. The authenticator can thus act as two kinds of authentication factor, which enables multi-factor authentication while eliminating the need to share a password with the Relying Party.

The four combinations not named in Table 2 have less distinguished use cases:

- The credential storage modality is less relevant for a platform authenticator than for a roaming authenticator, since users using a platform authenticator can typically be identified by a session cookie or the like (i.e., ambient credentials).
- A roaming authenticator that is discoverable credential capable but not multi-factor capable can be used for single-factor authentication without a username, where the user is automatically identified by the user handle and possession of the credential private key is used as the only authentication factor. This can be useful in some situations, but makes the user particularly vulnerable to theft of the authenticator.
- A roaming authenticator that is multi-factor capable but not discoverable credential capable can be used for multi-factor authentication, but requires the user to be identified first which risks leaking personally identifying information; see § 14.6.3 Privacy leak via credential IDs.

The following subsections define the aspects authenticator attachment modality, credential storage modality and authentication factor capability in more depth.

§ 6.2.1. *Authenticator Attachment Modality*

Clients can communicate with authenticators using a variety of mechanisms. For example, a client MAY use a client device-specific API to communicate with an authenticator which is physically bound to a client device. On the other hand, a client can use a variety of standardized cross-platform transport protocols such as Bluetooth (see § 5.8.4 Authenticator Transport Enumeration (enum AuthenticatorTransport)) to discover and communicate with cross-platform attached authenticators. We refer to authenticators that are part of the client device as ***platform authenticators***, while those that are reachable via cross-platform transport protocols are referred to as ***roaming authenticators***.

- A platform authenticator is attached using a client device-specific transport, called ***platform attachment***, and is usually not removable from the client device. A public key credential bound to a platform authenticator is called a ***platform credential***.

- A roaming authenticator is attached using cross-platform transports, called ***cross-platform attachment***. Authenticators of this class are removable from, and can "roam" between, client devices. A public key credential bound to a roaming authenticator is called a ***roaming credential***.

Some platform authenticators could possibly also act as roaming authenticators depending on context. For example, a platform authenticator integrated into a mobile device could make itself available as a roaming authenticator via Bluetooth. In this case clients running on the mobile device would recognise the authenticator as a platform authenticator, while clients running on a different client device and communicating with the same authenticator via Bluetooth would recognize it as a roaming authenticator.

The primary use case for platform authenticators is to register a particular client device as a "trusted device", so the client device itself acts as a something you have authentication factor for future authentication. This gives the user the convenience benefit of not needing a roaming authenticator for future authentication ceremonies, e.g., the user will not have to dig around in their pocket for their key fob or phone.

Use cases for roaming authenticators include: authenticating on a new client device for the first time, on rarely used client devices, client devices shared between multiple users, or client devices that do not include a platform authenticator; and when policy or preference dictates that the authenticator be kept separate from the client devices it is used with. A roaming authenticator can also be used to hold backup credentials in case another authenticator is lost.

§ 6.2.2. Credential Storage Modality

An authenticator can store a public key credential source in one of two ways:

1. In persistent storage embedded in the authenticator, client or client device, e.g., in a secure element. This is a technical requirement for a client-side discoverable public key credential source.
2. By encrypting (i.e., wrapping) the credential private key such that only this authenticator can decrypt (i.e., unwrap) it and letting the resulting ciphertext be the credential ID for the public key credential source. The credential ID is stored

by the Relying Party and returned to the authenticator via the `allowCredentials` option of `get()`, which allows the authenticator to decrypt and use the credential private key.

This enables the authenticator to have unlimited storage capacity for credential private keys, since the encrypted credential private keys are stored by the Relying Party instead of by the authenticator - but it means that a credential stored in this way must be retrieved from the Relying Party before the authenticator can use it.

Which of these storage strategies an authenticator supports defines the authenticator's ***credential storage modality*** as follows:

- An authenticator has the ***client-side credential storage modality*** if it supports client-side discoverable public key credential sources. An authenticator with client-side credential storage modality is also called ***discoverable credential capable***.
- An authenticator has the ***server-side credential storage modality*** if it does not have the client-side credential storage modality, i.e., it only supports storing credential private keys as a ciphertext in the credential ID.

Note that a discoverable credential capable authenticator MAY support both storage strategies. In this case, the authenticator MAY at its discretion use different storage strategies for different credentials, though subject to the `residentKey` or `requireResidentKey` options of `create()`.

§ 6.2.3. *Authentication Factor Capability*

There are three broad classes of authentication factors that can be used to prove an identity during an authentication ceremony: something you have, something you know and something you are. Examples include a physical key, a password, and a fingerprint, respectively.

All WebAuthn Authenticators belong to the something you have class, but an authenticator that supports user verification can also act as one or two additional kinds of authentication factor. For example, if the authenticator can verify a PIN, the PIN is something you know, and a biometric authenticator can verify something you are. Therefore, an authenticator that

supports user verification is ***multi-factor capable***. Conversely, an authenticator that is not multi-factor capable is ***single-factor capable***. Note that a single multi-factor capable authenticator could support several modes of user verification, meaning it could act as all three kinds of authentication factor.

Although user verification is performed locally on the authenticator and not by the Relying Party, the authenticator indicates if user verification was performed by setting the UV flag in the signed response returned to the Relying Party. The Relying Party can therefore use the UV flag to verify that additional authentication factors were used in a registration or authentication ceremony. The authenticity of the UV flag can in turn be assessed by inspecting the authenticator's attestation statement.

6.3. *Authenticator Operations*

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

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

6.3.1. Lookup Credential Source by Credential ID Algorithm

The result of ***looking up*** a credential id *credentialId* in an authenticator *authenticator* is the result of the following algorithm:

1. If *authenticator* can decrypt *credentialId* into a public key credential source *credSource*:
    1. Set *credSource*.id to *credentialId*.
    2. Return *credSource*.

2. For each public key credential source *credSource* of *authenticator*'s credentials map:

    1. If *credSource*.id is *credentialId*, return *credSource*.

3. Return `null`.

§ 6.3.2. The *authenticatorMakeCredential* Operation

It takes the following input parameters:

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

rpEntity
: The Relying Party's `PublicKeyCredentialRpEntity`.

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

requireResidentKey
: The effective resident key requirement for credential creation, a Boolean value determined by the client.

requireUserPresence
: The constant Boolean value `true`. It is included here as a pseudo-parameter to simplify applying this abstract authenticator model to implementations that may wish to make a test of user presence optional although WebAuthn does not.

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

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

excludeCredentialDescriptorList

An OPTIONAL list of `PublicKeyCredentialDescriptor` objects provided by the Relying Party with the intention that, if any of these are known to the authenticator, it SHOULD NOT create a new credential. *excludeCredentialDescriptorList* contains a list of known credentials.

enterpriseAttestationPossible

A Boolean value that indicates that individually-identifying attestation MAY be returned by the authenticator.

extensions

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

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

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

1. Check if all the supplied parameters are syntactically well-formed and of the correct length. If not, return an error code equivalent to "`UnknownError`" and terminate the operation.
2. Check if at least one of the specified combinations of `PublicKeyCredentialType` and cryptographic parameters in *credTypesAndPubKeyAlgs* is supported. If not, return an error code equivalent to "`NotSupportedError`" and terminate the operation.
3. For each *descriptor* of *excludeCredentialDescriptorList*:
    1. If looking up *`descriptor.`*`id` in this authenticator returns non-null, and the returned item's RP ID and type match *`rpEntity.`*`id` and *`excludeCredentialDescriptorList.`*`type` respectively, then collect an authorization gesture confirming user consent for creating a new credential. The authorization gesture MUST include a test of user presence. If the user

↪ **confirms consent to create a new credential**

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

↪ **does not consent to create a new credential**

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

> Note: The purpose of this authorization gesture is not to proceed with creating a credential, but for privacy reasons to authorize disclosure of the fact that *`descriptor.id`* is bound to this authenticator. If the user consents, the client and Relying Party can detect this and guide the user to use a different authenticator. If the user does not consent, the authenticator does not reveal that *`descriptor.id`* is bound to it, and responds as if the user simply declined consent to create a credential.

4. If *requireResidentKey* is `true` and the authenticator cannot store a client-side discoverable public key credential source, return an error code equivalent to "`ConstraintError`" and terminate the operation.

5. If *requireUserVerification* is `true` and the authenticator cannot perform user verification, return an error code equivalent to "`ConstraintError`" and terminate the operation.

¶ 6. Collect an authorization gesture confirming user consent for creating a new credential. The prompt for the authorization gesture is shown by the authenticator if it has its own output capability, or by the user agent otherwise. The prompt SHOULD display *`rpEntity.id`*, *`rpEntity.name`*, *`userEntity.name`* and *`userEntity.displayName`*, if possible.

   If *requireUserVerification* is `true`, the authorization gesture MUST include user verification.

   If *requireUserPresence* is `true`, the authorization gesture MUST include a test of user presence.

   If the user does not consent or if user verification fails, return an error code equivalent to "`NotAllowedError`" and terminate the operation.

7. Once the authorization gesture has been completed and user consent has been obtained, generate a new credential object:

1. Let (*publicKey*, *privateKey*) be a new pair of cryptographic keys using the combination of `PublicKeyCredentialType` and cryptographic parameters represented by the first item in *credTypesAndPubKeyAlgs* that is supported by this authenticator.

2. Let *userHandle* be *`userEntity.`*`id`.

3. Let *credentialSource* be a new public key credential source with the fields:

   type
   : `public-key`.

   privateKey
   : *privateKey*

   rpId
   : *`rpEntity.`*`id`

   userHandle
   : *userHandle*

   otherUI
   : Any other information the authenticator chooses to include.

4. If *requireResidentKey* is `true` or the authenticator chooses to create a client-side discoverable public key credential source:

   1. Let *credentialId* be a new credential id.

   2. Set *credentialSource*.id to *credentialId*.

   3. Let *credentials* be this authenticator's credentials map.

   4. Set *credentials*[(*`rpEntity.`*`id`, *userHandle*)] to *credentialSource*.

5. Otherwise:

1. Let *credentialId* be the result of serializing and encrypting *credentialSource* so that only this authenticator can decrypt it.

8. If any error occurred while creating the new credential object, return an error code equivalent to "`UnknownError`" and terminate the operation.

9. Let *processedExtensions* be the result of authenticator extension processing for each supported extension identifier → authenticator extension input in *extensions*.

10. If the authenticator:

    ↪ **is a U2F device**
    let the signature counter value for the new credential be zero. (U2F devices may support signature counters but do not return a counter when making a credential. See [FIDO-U2F-Message-Formats].)

    ↪ **supports a global signature counter**
    Use the global signature counter's actual value when generating authenticator data.

    ↪ **supports a per credential signature counter**
    allocate the counter, associate it with the new credential, and initialize the counter value as zero.

    ↪ **does not support a signature counter**
    let the signature counter value for the new credential be constant at zero.

11. Let *attestedCredentialData* be the attested credential data byte array including the *credentialId* and *publicKey*.

12. Let *authenticatorData* be the byte array specified in § 6.1 Authenticator Data, including *attestedCredentialData* as the `attestedCredentialData` and *processedExtensions*, if any, as the `extensions`.

13. Create an attestation object for the new credential using the procedure specified in § 6.5.4 Generating an Attestation Object, using an authenticator-chosen attestation statement format, *authenticatorData*, and *hash*, as well as `taking into account` the value of *enterpriseAttestationPossible*. For more details on attestation, see § 6.5 Attestation.

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

§ 6.3.3. The *authenticatorGetAssertion* Operation

It takes the following input parameters:

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

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

allowCredentialDescriptorList
: An OPTIONAL list of `PublicKeyCredentialDescriptor`s describing credentials acceptable to the Relying Party (possibly filtered by the client), if any.

requireUserPresence
: The constant Boolean value `true`. It is included here as a pseudo-parameter to simplify applying this abstract authenticator model to implementations that may wish to make a test of user presence optional although WebAuthn does not.

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

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

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

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

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

2. Let *credentialOptions* be a new empty set of public key credential sources.
3. If *allowCredentialDescriptorList* was supplied, then for each *descriptor* of *allowCredentialDescriptorList*:
    1. Let *credSource* be the result of looking up `descriptor.id` in this authenticator.
    2. If *credSource* is not `null`, append it to *credentialOptions*.
4. Otherwise (*allowCredentialDescriptorList* was not supplied), for each *key* → *credSource* of this authenticator's credentials map, append *credSource* to *credentialOptions*.
5. Remove any items from *credentialOptions* whose rpId is not equal to *rpId*.
6. If *credentialOptions* is now empty, return an error code equivalent to "`NotAllowedError`" and terminate the operation.
7. Prompt the user to select a public key credential source *selectedCredential* from *credentialOptions*. Collect an authorization gesture confirming user consent for using *selectedCredential*. The prompt for the authorization gesture may be shown by the authenticator if it has its own output capability, or by the user agent otherwise.

    If *requireUserVerification* is `true`, the authorization gesture MUST include user verification.

    If *requireUserPresence* is `true`, the authorization gesture MUST include a test of user presence.

    If the user does not consent, return an error code equivalent to "`NotAllowedError`" and terminate the operation.
8. Let *processedExtensions* be the result of authenticator extension processing for each supported extension identifier → authenticator extension input in *extensions*.
9. Increment the credential associated signature counter or the global signature counter value, depending on which approach is implemented by the authenticator, by some positive value. If the authenticator does not implement a signature counter, let the signature counter value remain constant at zero.
10. Let *authenticatorData* be the byte array specified in § 6.1 Authenticator Data including *processedExtensions*, if any, as the `extensions` and excluding `attestedCredentialData`.

11. Let *signature* be the assertion signature of the concatenation `authenticatorData || hash` using the privateKey of *selectedCredential* as shown in Figure 4, below. A simple, undelimited concatenation is safe to use here because the authenticator data describes its own length. The hash of the serialized client data (which potentially has a variable length) is always the last element.

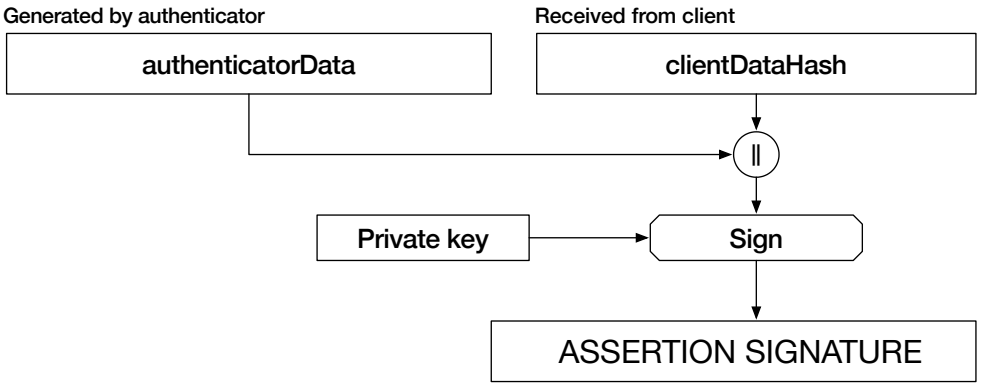

Figure 4 *Generating an assertion signature.*

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

¶ 13. Return to the user agent:

- *selectedCredential*.id, if either a list of credentials (i.e., *allowCredentialDescriptorList*) of length 2 or greater was supplied by the client, or no such list was supplied.

  > Note: If, within *allowCredentialDescriptorList*, the client supplied exactly one credential and it was successfully employed, then its credential ID is not returned since the client already knows it. This saves transmitting these bytes over what may be a constrained connection in what is likely a common case.

- *authenticatorData*

- *signature*
- *selectedCredential*.userHandle

> Note: the returned userHandle value may be `null`, see: userHandleResult.

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

6.3.4. The *authenticatorCancel* Operation

This operation takes no input parameters and returns no result.

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

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

6.4. String Handling

Authenticators may be required to store arbitrary strings chosen by a Relying Party, for example the `name` and `displayName` in a `PublicKeyCredentialUserEntity`. This section discusses some practical consequences of handling arbitrary strings that may be presented to humans.

§ 6.4.1. String Truncation

Each arbitrary string in the API will have some accommodation for the potentially limited resources available to an authenticator. If string value truncation is the chosen accommodation then authenticators MAY truncate in order to make the string fit within a length equal or greater than the specified minimum supported length. Such truncation SHOULD also respect UTF-8 sequence boundaries or grapheme cluster boundaries [UTR29]. This defines the maximum truncation permitted and authenticators MUST NOT truncate further.

For example, in figure 5 the string is 65 bytes long. If truncating to 64 bytes then the final 0x88 byte must be removed purely because of space reasons. Since that leaves a partial UTF-8 sequence the remainder of that sequence may also be removed. Since that leaves a partial grapheme cluster an authenticator may remove the remainder of that cluster.

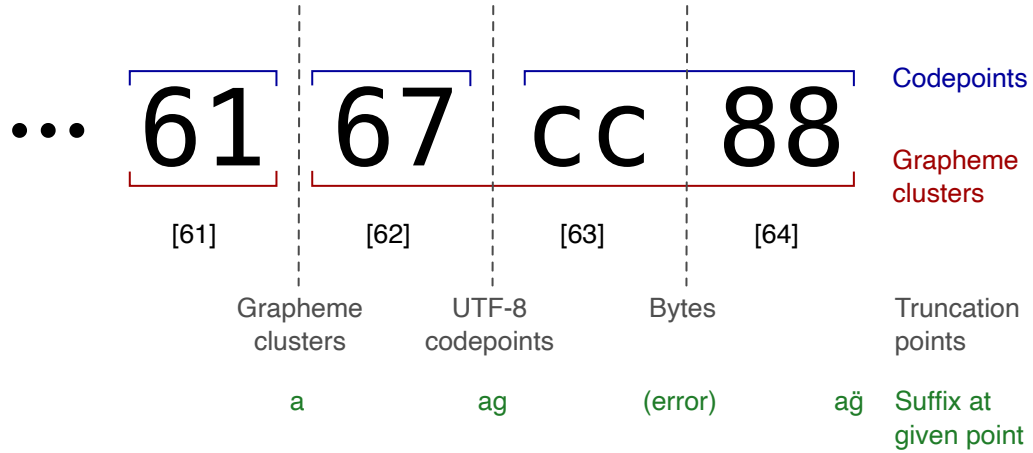

Figure 5 *The end of a UTF-8 encoded string showing the positions of different truncation boundaries.*

Conforming User Agents are responsible for ensuring that the authenticator behaviour observed by Relying Parties conforms to this specification with respect to string handling. For example, if an authenticator is known to behave incorrectly when asked to store large strings, the user agent SHOULD perform the truncation for it in order to maintain the model from the point of view of the Relying Party. User-agents that do this SHOULD truncate at grapheme cluster boundaries.

Truncation based on UTF-8 sequences alone may cause a grapheme cluster to be truncated. This could make the grapheme cluster render as a different glyph, potentially changing the meaning of the string, instead of removing the glyph entirely.

In addition to that, truncating on byte boundaries alone causes a known issue that user agents should be aware of: if the authenticator is using [FIDO-CTAP] then future messages from the authenticator may contain invalid CBOR since the value is typed as a CBOR string and thus is required to be valid UTF-8. User agents are tasked with handling this to avoid burdening authenticators with understanding character encodings and Unicode character properties. Thus, when dealing with authenticators, user agents SHOULD:

1. Ensure that any strings sent to authenticators are validly encoded.
2. Handle the case where strings have been truncated resulting in an invalid encoding. For example, any partial code point at the end may be dropped or replaced with U+FFFD.

§ 6.4.2. Language and Direction Encoding

In order to be correctly displayed in context, the language and base direction of a string may be required. Strings in this API may have to be written to fixed-function authenticators and then later read back and displayed on a different platform. Thus language and direction metadata is encoded in the string itself to ensure that it is transported atomically.

To encode language and direction metadata in a string that is documented as permitting it, suffix its code points with two sequences of code points:

The first encodes a language tag with the code point U+E0001 followed by the ASCII values of the language tag each shifted up by U+E0000. For example, the language tag “en-US” becomes the code points U+E0001, U+E0065, U+E006E, U+E002D, U+E0055, U+E0053.

The second consists of a single code point which is either U+200E (“LEFT-TO-RIGHT MARK”), U+200F (“RIGHT-TO-LEFT MARK”), or U+E007F (“CANCEL TAG”). The first two can be used to indicate directionality but SHOULD only be

used when neccessary to produce the correct result. (E.g. an RTL string that starts with LTR-strong characters.) The value U+E007F is a direction-agnostic indication of the end of the language tag.

So the string "حبيب الرحمان" could have two different DOMString values, depending on whether the language was encoded or not. (Since the direction is unambigous a directionality marker is not needed in this example.)

- Unadorned string: U+FEA2, U+FE92, U+FBFF, U+FE91, U+20, U+FE8E, U+FEDF, U+FEAE, U+FEA4, U+FEE3, U+FE8E, U+FEE7
- With language "ar-SA" encoded: U+FEA2, U+FE92, U+FBFF, U+FE91, U+20, U+FE8E, U+FEDF, U+FEAE, U+FEA4, U+FEE3, U+FE8E, U+FEE7, U+E0001, U+E0061, U+E0072, U+E002D, U+E0053, U+E0041, U+E007F

Consumers of strings that may have language and direction encoded should be aware that truncation could truncate a language tag into a different, but still valid, language. The final directionality marker or CANCEL TAG code point provide an unambigous indication of truncation.

§ 6.5. Attestation

Authenticators SHOULD also provide some form of attestation, if possible. If an authenticator does, the basic requirement is that the authenticator can produce, for each credential public key, an attestation statement verifiable by the WebAuthn Relying Party. Typically, this attestation statement contains a signature by an attestation private key over the attested credential public key and a challenge, as well as a certificate or similar data providing provenance information for the attestation public key, enabling the Relying Party to make a trust decision. However, if an attestation key pair is not available, then the authenticator MAY either perform self attestation of the credential public key with the corresponding credential private key, or otherwise perform no attestation. All this information is returned by authenticators any time a new public key credential is generated, in the overall form of an ***attestation object***. The relationship of the attestation object with authenticator data (containing attested credential data) and the attestation statement is illustrated in figure 6, below.

If an authenticator employs self attestation or no attestation, then no provenance information is provided for the Relying Party to base a trust decision on. In these cases, the authenticator provides no guarantees about its operation to the Relying Party.

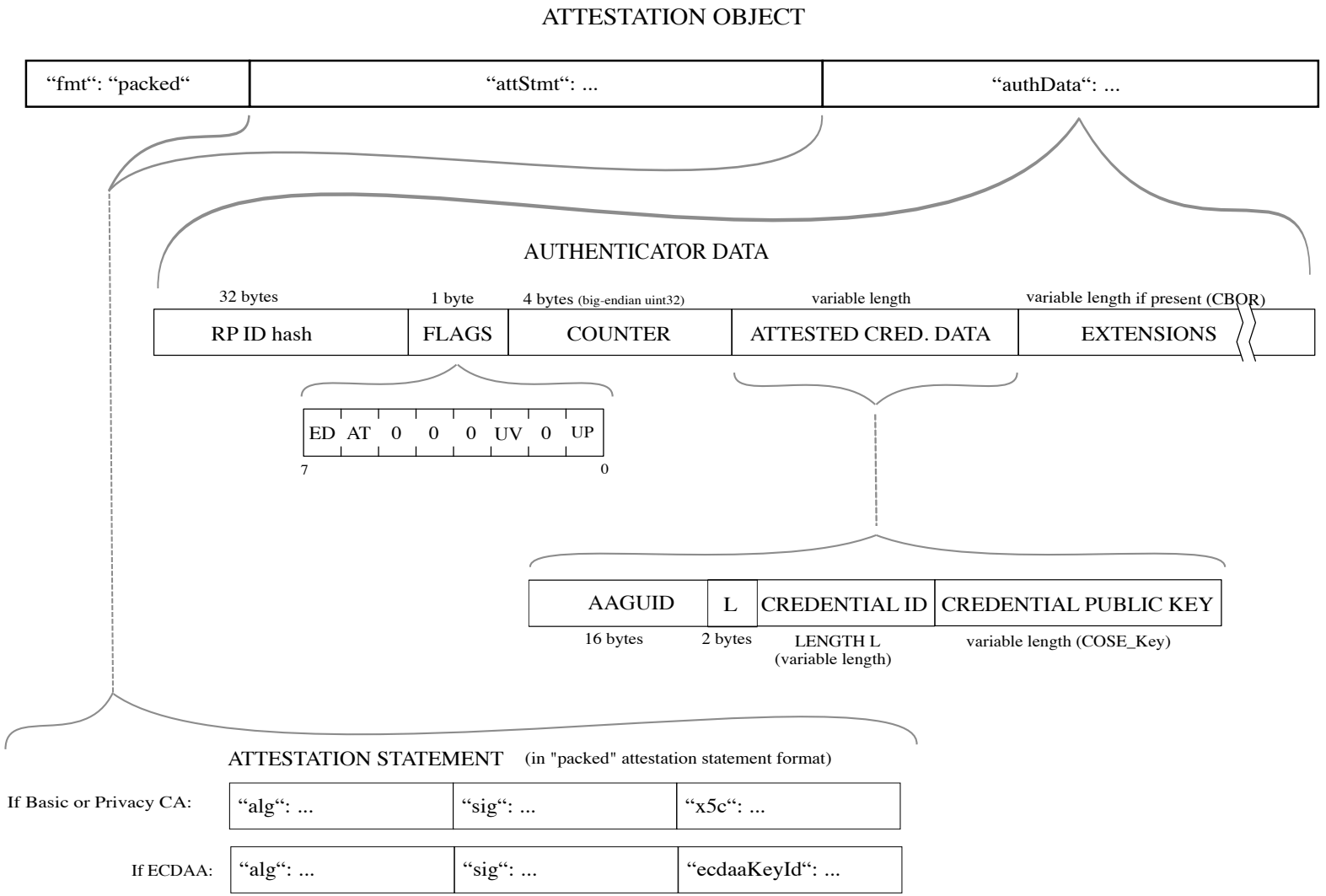

Figure 6 *Attestation object layout illustrating the included authenticator data (containing attested credential data) and the attestation statement.*

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

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

1. The ***attestation statement format*** is the manner in which the signature is represented and the various contextual bindings are incorporated into the attestation statement by the authenticator. In other words, this defines the syntax of the statement. Various existing components and OS platforms (such as TPMs and the Android OS) have previously defined attestation statement formats. This specification supports a variety of such formats in an extensible way, as defined in § 6.5.2 Attestation Statement Formats. The formats themselves are identified by strings, as described in § 8.1 Attestation Statement Format Identifiers.
2. The ***attestation type*** defines the semantics of attestation statements and their underlying trust models. Specifically, it defines how a Relying Party establishes trust in a particular attestation statement, after verifying that it is cryptographically valid. This specification supports a number of attestation types, as described in § 6.5.3 Attestation Types.

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

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

- The attestation type, which determines the trust model,

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

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

§ 6.5.1. Attested Credential Data

Attested credential data is a variable-length byte array added to the authenticator data when generating an attestation object for a given credential. Its format is shown in Table 3.

Name	Length (in bytes)	Description
aaguid	16	The AAGUID of the authenticator.
credentialIdLength	2	Byte length **L** of Credential ID, 16-bit unsigned big-endian integer.
credentialId	L	Credential ID
credentialPublicKey	variable	The credential public key encoded in COSE_Key format, as defined in Section 7 of [RFC8152], using the CTAP2 canonical CBOR encoding form. The COSE_Key-encoded credential public key MUST contain the "alg" parameter and MUST NOT contain any other OPTIONAL parameters. The "alg" parameter MUST contain a `COSEAlgorithmIdentifier` value. The encoded credential public key MUST also contain any additional REQUIRED parameters stipulated by the relevant key type specification, i.e., REQUIRED for the key type "kty" and algorithm "alg" (see Section 8 of [RFC8152]).

Table 3 *Attested credential data layout. The names in the Name column are only for reference within this document, and are not present in the actual representation of the attested credential data.*

§ *6.5.1.1. Examples of `credentialPublicKey` Values Encoded in COSE_Key Format*

This section provides examples of COSE_Key-encoded Elliptic Curve and RSA public keys for the ES256, PS256, and RS256 signature algorithms. These examples adhere to the rules defined above for the credentialPublicKey value, and are presented in CDDL [RFC8610] for clarity.

[RFC8152] Section 7 defines the general framework for all COSE_Key-encoded keys. Specific key types for specific algorithms are defined in other sections of [RFC8152] as well as in other specifications, as noted below.

Below is an example of a COSE_Key-encoded Elliptic Curve public key in EC2 format (see [RFC8152] Section 13.1), on the P-256 curve, to be used with the ES256 signature algorithm (ECDSA w/ SHA-256, see [RFC8152] Section 8.1:

```
EXAMPLE 6
{
  1:   2,  ; kty: EC2 key type
  3:  -7,  ; alg: ES256 signature algorithm
 -1:   1,  ; crv: P-256 curve
 -2:   x,  ; x-coordinate as byte string 32 bytes in length
           ; e.g., in hex: 65eda5a12577c2bae829437fe338701a10aaa375e1bb5b5de108de439c08551
 -3:   y   ; y-coordinate as byte string 32 bytes in length
           ; e.g., in hex: 1e52ed75701163f7f9e40ddf9f341b3dc9ba860af7e0ca7ca7e9eecd0084d19
}
```

Below is the above Elliptic Curve public key encoded in the CTAP2 canonical CBOR encoding form, whitespace and line breaks are included here for clarity and to match the CDDL [RFC8610] presentation above:

```
EXAMPLE 7
A5
   01  02

   03  26

   20  01

   21  58 20   65eda5a12577c2bae829437fe338701a10aaa375e1bb5b5de108de439c08551d

   22  58 20   1e52ed75701163f7f9e40ddf9f341b3dc9ba860af7e0ca7ca7e9eecd0084d19c
```

Below is an example of a COSE_Key-encoded 2048-bit RSA public key (see [RFC8230] Section 4, to be used with the PS256 signature algorithm (RSASSA-PSS with SHA-256, see [RFC8230] Section 2:

```
EXAMPLE 8
{
  1:   3,  ; kty: RSA key type
  3: -37,  ; alg: PS256
 -1:   n,  ; n:   RSA modulus n byte string 256 bytes in length
           ;      e.g., in hex (middle bytes elided for brevity): DB5F651550...6DC6548ACC3
 -2:   e   ; e:   RSA public exponent e byte string 3 bytes in length
           ;      e.g., in hex: 010001
}
```

Below is an example of the same COSE_Key-encoded RSA public key as above, to be used with the RS256 signature algorithm (RSASSA-PKCS1-v1_5 with SHA-256):

```
EXAMPLE 9
{
  1:   3,  ; kty: RSA key type
  3:-257,  ; alg: RS256
 -1:   n,  ; n:   RSA modulus n byte string 256 bytes in length
           ;      e.g., in hex (middle bytes elided for brevity): DB5F651550...6DC6548ACC3
 -2:   e   ; e:   RSA public exponent e byte string 3 bytes in length
           ;      e.g., in hex: 010001
}
```

§ 6.5.2. Attestation Statement Formats

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

- **Attestation statement format identifier:**
- **Supported attestation types:**
- **Syntax:** The syntax of an attestation statement produced in this format, defined using CDDL [RFC8610] for the extension point `$attStmtFormat` defined in § 6.5.4 Generating an Attestation Object.
- ***Signing procedure***: The signing procedure for computing an attestation statement in this format given the public key credential to be attested, the authenticator data structure containing the ***authenticator data for the attestation***, and the hash of the serialized client data.

- ***Verification procedure***: The procedure for verifying an attestation statement, which takes the following ***verification procedure inputs***:

  - *attStmt*: The attestation statement structure
  - *authenticatorData*: The ***authenticator data*** ***claimed to have been used for the attestation***
  - *clientDataHash*: The hash of the serialized client data

  The procedure returns either:

  - An error indicating that the attestation is invalid, or
  - An implementation-specific value representing the attestation type, and the trust path. This ***attestation trust path*** is either empty (in case of self attestation), or a set of X.509 certificates.

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

§ 6.5.3. Attestation Types

WebAuthn supports several attestation types, defining the semantics of attestation statements and their underlying trust models:

Note: This specification does not define any data structures explicitly expressing the attestation types employed by authenticators. Relying Parties engaging in attestation statement verification — i.e., when calling `navigator.credentials.create()` they select an attestation conveyance other than `none` and verify the received attestation statement — will determine the employed attestation type as a part of verification. See the "Verification procedure" subsections of § 8 Defined Attestation Statement Formats. See also § 14.4.1 Attestation Privacy. For all attestation types defined in this section other than Self and None, Relying Party verification is followed by matching the trust path to an acceptable root certificate per step 21 of § 7.1 Registering a New Credential. Differentiating these attestation types becomes useful primarily as a means for determining if the attestation is acceptable under Relying Party policy.

***Basic Attestation* (*Basic*)**

In the case of basic attestation [UAFProtocol], the authenticator's attestation key pair is specific to an authenticator "model", i.e., a "batch" of authenticators. Thus, authenticators of the same, or similar, model often share the same attestation key pair. See § 14.4.1 Attestation Privacy for further information.

Basic attestation is also referred to as ***batch attestation***.

***Self Attestation* (*Self*)**

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

***Attestation CA* (*AttCA*)**

In this case, an authenticator is based on a Trusted Platform Module (TPM) and holds an authenticator-specific "endorsement key" (EK). This key is used to securely communicate with a trusted third party, the Attestation CA [TCG-CMCProfile-AIKCertEnroll] (formerly known as a "Privacy CA"). The authenticator can generate multiple attestation identity key pairs (AIK) and requests an Attestation CA to issue an AIK certificate for each. Using this approach, such an authenticator can limit the exposure of the EK (which is a global correlation handle) to Attestation CA(s). AIKs can be requested for each authenticator-generated public key credential individually, and conveyed to Relying Parties as attestation certificates.

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

***Anonymization CA* (*AnonCA*)**

In this case, the authenticator uses an Anonymization CA which dynamically generates per-credential attestation certificates such that the attestation statements presented to Relying Parties do not provide uniquely identifiable information, e.g., that might be used for tracking purposes.

> Note: Attestation statements conveying attestations of type AttCA or AnonCA use the same data structure as those of type Basic, so the three attestation types are, in general, distinguishable only with externally provided knowledge regarding the contents of the attestation certificates conveyed in the attestation statement.

No attestation statement (*None*)

In this case, no attestation information is available. See also § 8.7 None Attestation Statement Format.

§ 6.5.4. Generating an Attestation Object

To generate an attestation object (see: Figure 6) given:

attestationFormat

An attestation statement format.

authData

A byte array containing authenticator data.

hash

The hash of the serialized client data.

the authenticator MUST:

1. Let *attStmt* be the result of running *attestationFormat*'s signing procedure given *authData* and *hash*.

2. Let *fmt* be *attestationFormat*'s attestation statement format identifier

3. Return the attestation object as a CBOR map with the following syntax, filled in with variables initialized by this algorithm:

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

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

; Every attestation statement format must have the above fields
attStmtTemplate .within $$attStmtType
```

§ 6.5.5. Signature Formats for Packed Attestation, FIDO U2F Attestation, and Assertion Signatures

- For COSEAlgorithmIdentifier -7 (ES256), and other ECDSA-based algorithms, the `sig` value MUST be encoded as an ASN.1 DER Ecdsa-Sig-Value, as defined in [RFC3279] section 2.2.3.

```
Example:
30 44                                   ; SEQUENCE (68 Bytes)
    02 20                               ; INTEGER (32 Bytes)
    |  3d 46 28 7b 8c 6e 8c 8c  26 1c 1b 88 f2 73 b0 9a
    |  32 a6 cf 28 09 fd 6e 30  d5 a7 9f 26 37 00 8f 54
    02 20                               ; INTEGER (32 Bytes)
    |  4e 72 23 6e a3 90 a9 a1  7b cf 5f 7a 09 d6 3a b2
    |  17 6c 92 bb 8e 36 c0 41  98 a2 7b 90 9b 6e 8f 13
```

> Note: As CTAP1/U2F authenticators are already producing signatures values in this format, CTAP2 authenticators will also produce signatures values in the same format, for consistency reasons.

It is RECOMMENDED that any new attestation formats defined not use ASN.1 encodings, but instead represent signatures as equivalent fixed-length byte arrays without internal structure, using the same representations as used by COSE signatures as defined in [RFC8152] and [RFC8230].

The below signature format definitions satisfy this requirement and serve as examples for deriving the same for other signature algorithms not explicitly mentioned here:

- For COSEAlgorithmIdentifier -257 (RS256), `sig` MUST contain the signature generated using the RSASSA-PKCS1-v1_5 signature scheme defined in section 8.2.1 in [RFC8017] with SHA-256 as the hash function. The signature is not ASN.1 wrapped.
- For COSEAlgorithmIdentifier -37 (PS256), `sig` MUST contain the signature generated using the RSASSA-PSS signature scheme defined in section 8.1.1 in [RFC8017] with SHA-256 as the hash function. The signature is not ASN.1 wrapped.

§ 7. WebAuthn Relying Party Operations

A registration or authentication ceremony begins with the WebAuthn Relying Party creating a `PublicKeyCredentialCreationOptions` or `PublicKeyCredentialRequestOptions` object, respectively, which encodes the parameters for the ceremony. The Relying Party SHOULD take care to not leak sensitive information during this stage; see § 14.6.2 Username Enumeration for details.

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

outside the scope of this specification. This section describes the operations that the Relying Party must perform upon receipt of these structures.

§ 7.1. Registering a New Credential

In order to perform a registration ceremony, the Relying Party MUST proceed as follows:

1. Let *options* be a new `PublicKeyCredentialCreationOptions` structure configured to the Relying Party's needs for the ceremony.
2. Call `navigator.credentials.create()` and pass *options* as the `publicKey` option. Let *credential* be the result of the successfully resolved promise. If the promise is rejected, abort the ceremony with a user-visible error, or otherwise guide the user experience as might be determinable from the context available in the rejected promise. For example if the promise is rejected with an error code equivalent to "`InvalidStateError`", the user might be instructed to use a different authenticator. For information on different error contexts and the circumstances leading to them, see § 6.3.2 The authenticatorMakeCredential Operation.
3. Let *response* be `credential.response`. If *response* is not an instance of `AuthenticatorAttestationResponse`, abort the ceremony with a user-visible error.
4. Let *clientExtensionResults* be the result of calling `credential.getClientExtensionResults()`.
5. Let *JSONtext* be the result of running UTF-8 decode on the value of `response.clientDataJSON`.

   > Note: Using any implementation of UTF-8 decode is acceptable as long as it yields the same result as that yielded by the UTF-8 decode algorithm. In particular, any leading byte order mark (BOM) MUST be stripped.

6. Let *C*, the client data claimed as collected during the credential creation, be the result of running an implementation-specific JSON parser on *JSONtext*.

> Note: *C* may be any implementation-specific data structure representation, as long as *C*'s components are referenceable, as required by this algorithm.

7. Verify that the value of *C*.`type` is `webauthn.create`.
8. Verify that the value of *C*.`challenge` equals the base64url encoding of *options*.`challenge`.
9. Verify that the value of *C*.`origin` matches the Relying Party's origin.
10. Verify that the value of *C*.`tokenBinding`.`status` matches the state of Token Binding for the TLS connection over which the assertion was obtained. If Token Binding was used on that TLS connection, also verify that *C*.`tokenBinding`.`id` matches the base64url encoding of the Token Binding ID for the connection.
11. Let *hash* be the result of computing a hash over *response*.`clientDataJSON` using SHA-256.
12. 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*.
13. Verify that the `rpIdHash` in *authData* is the SHA-256 hash of the RP ID expected by the Relying Party.
14. Verify that the User Present bit of the `flags` in *authData* is set.
15. If user verification is required for this registration, verify that the User Verified bit of the `flags` in *authData* is set.
16. Verify that the "alg" parameter in the credential public key in *authData* matches the `alg` attribute of one of the items in *options*.`pubKeyCredParams`.
17. Verify that the values of the client extension outputs in *clientExtensionResults* and the authenticator extension outputs in the `extensions` in *authData* are as expected, considering the client extension input values that were given in *options*.`extensions` and any specific policy of the Relying Party regarding unsolicited extensions, i.e., those that were not specified as part of *options*.`extensions`. In the general case, the meaning of "are as expected" is specific to the Relying Party and which extensions are in use.

> Note: Client platforms MAY enact local policy that sets additional authenticator extensions or client extensions and thus cause values to appear in the authenticator extension outputs or client extension outputs that were not originally specified as part of *`options`*`.extensions`. Relying Parties MUST be prepared to handle such situations, whether it be to ignore the unsolicited extensions or reject the attestation. The Relying Party can make this decision based on local policy and the extensions in use.

> Note: Since all extensions are OPTIONAL for both the client and the authenticator, the Relying Party MUST also be prepared to handle cases where none or not all of the requested extensions were acted upon.

18. Determine the attestation statement format by performing a USASCII case-sensitive match on *fmt* against the set of supported WebAuthn Attestation Statement Format Identifier values. An up-to-date list of registered WebAuthn Attestation Statement Format Identifier values is maintained in the IANA "WebAuthn Attestation Statement Format Identifiers" registry [IANA-WebAuthn-Registries] established by [RFC8809].

19. Verify that *attStmt* is a correct attestation statement, conveying a valid attestation signature, by using the attestation statement format *fmt*'s verification procedure given *attStmt*, *authData* and *hash*.

> Note: Each attestation statement format specifies its own verification procedure. See § 8 Defined Attestation Statement Formats for the initially-defined formats, and [IANA-WebAuthn-Registries] for the up-to-date list.

20. If validation is successful, obtain a list of acceptable trust anchors (i.e. attestation root certificates) for that attestation type and attestation statement format *fmt*, from a trusted source or from policy. For example, the FIDO Metadata Service [FIDOMetadataService] provides one way to obtain such information, using the `aaguid` in the `attestedCredentialData` in *authData*.

21. Assess the attestation trustworthiness using the outputs of the verification procedure in step 19, as follows:

    - If no attestation was provided, verify that None attestation is acceptable under Relying Party policy.
    - If self attestation was used, verify that self attestation is acceptable under Relying Party policy.

- Otherwise, use the X.509 certificates returned as the attestation trust path from the verification procedure to verify that the attestation public key either correctly chains up to an acceptable root certificate, or is itself an acceptable certificate (i.e., it and the root certificate obtained in Step 20 may be the same).

22. Check that the `credentialId` is not yet registered to any other user. If registration is requested for a credential that is already registered to a different user, the Relying Party SHOULD fail this registration ceremony, or it MAY decide to accept the registration, e.g. while deleting the older registration.

23. If the attestation statement *attStmt* verified successfully and is found to be trustworthy, then register the new credential with the account that was denoted in *`options`*`.user`:

    - Associate the user's account with the `credentialId` and `credentialPublicKey` in *`authData`*`.attestedCredentialData`, as appropriate for the Relying Party's system.
    - Associate the `credentialId` with a new stored signature counter value initialized to the value of *`authData`*`.signCount`.

    It is RECOMMENDED to also:

    - Associate the `credentialId` with the transport hints returned by calling *`credential`*`.response.getTransports()`. This value SHOULD NOT be modified before or after storing it. It is RECOMMENDED to use this value to populate the `transports` of the `allowCredentials` option in future `get()` calls to help the client know how to find a suitable authenticator.

24. If the attestation statement *attStmt* successfully verified but is not trustworthy per step 21 above, the Relying Party SHOULD fail the registration ceremony.

> NOTE: However, if permitted by policy, the Relying Party MAY register the credential ID and credential public key but treat the credential as one with self attestation (see § 6.5.3 Attestation Types). If doing so, the Relying Party is asserting there is no cryptographic proof that the public key credential has been generated by a particular authenticator model. See [FIDOSecRef] and [UAFProtocol] for a more detailed discussion.

Verification of attestation objects requires that the Relying Party has a trusted method of determining acceptable trust anchors in step 20 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.

§ 7.2. Verifying an Authentication Assertion

In order to perform an authentication ceremony, the Relying Party MUST proceed as follows:

1. Let *options* be a new `PublicKeyCredentialRequestOptions` structure configured to the Relying Party's needs for the ceremony.

   If *`options`*`.allowCredentials` is present, the `transports` member of each item SHOULD be set to the value returned by *`credential`*`.response.getTransports()` when the corresponding credential was registered.

2. Call `navigator.credentials.get()` and pass *options* as the `publicKey` option. Let *credential* be the result of the successfully resolved promise. If the promise is rejected, abort the ceremony with a user-visible error, or otherwise guide the user experience as might be determinable from the context available in the rejected promise. For information on different error contexts and the circumstances leading to them, see § 6.3.3 The authenticatorGetAssertion Operation.

3. Let *response* be *`credential`*`.response`. If *response* is not an instance of `AuthenticatorAssertionResponse`, abort the ceremony with a user-visible error.

4. Let *clientExtensionResults* be the result of calling *`credential`*`.getClientExtensionResults()`.

5. If *`options`*`.allowCredentials` is not empty, verify that *`credential`*`.id` identifies one of the public key credentials listed in *`options`*`.allowCredentials`.

6. Identify the user being authenticated and verify that this user is the owner of the public key credential source *credentialSource* identified by *`credential`*`.id`:

↪ **If the user was identified before the authentication ceremony was initiated, e.g., via a username or cookie,**
verify that the identified user is the owner of *credentialSource*. If `response.userHandle` is present, let *userHandle* be its value. Verify that *userHandle* also maps to the same user.

↪ **If the user was not identified before the authentication ceremony was initiated,**
verify that `response.userHandle` is present, and that the user identified by this value is the owner of *credentialSource*.

7. Using `credential.id` (or `credential.rawId`, if base64url encoding is inappropriate for your use case), look up the corresponding credential public key and let *credentialPublicKey* be that credential public key.
8. Let *cData*, *authData* and *sig* denote the value of *response*'s `clientDataJSON`, `authenticatorData`, and `signature` respectively.
9. Let *JSONtext* be the result of running UTF-8 decode on the value of *cData*.

> Note: Using any implementation of UTF-8 decode is acceptable as long as it yields the same result as that yielded by the UTF-8 decode algorithm. In particular, any leading byte order mark (BOM) MUST be stripped.

10. Let *C*, the client data claimed as used for the signature, be the result of running an implementation-specific JSON parser on *JSONtext*.

> Note: *C* may be any implementation-specific data structure representation, as long as *C*'s components are referenceable, as required by this algorithm.

11. Verify that the value of `C.type` is the string `webauthn.get`.
12. Verify that the value of `C.challenge` equals the base64url encoding of `options.challenge`.
13. Verify that the value of `C.origin` matches the Relying Party's origin.
14. Verify that the value of `C.tokenBinding.status` matches the state of Token Binding for the TLS connection over which the attestation was obtained. If Token Binding was used on that TLS connection, also verify that

`C.tokenBinding.id` matches the base64url encoding of the Token Binding ID for the connection.

15. Verify that the `rpIdHash` in *authData* is the SHA-256 hash of the RP ID expected by the Relying Party.

> Note: If using the appid extension, this step needs some special logic. See § 10.1 FIDO AppID Extension (appid) for details.

16. Verify that the User Present bit of the `flags` in *authData* is set.
17. If user verification is required for this assertion, verify that the User Verified bit of the `flags` in *authData* is set.
18. Verify that the values of the client extension outputs in *clientExtensionResults* and the authenticator extension outputs in the `extensions` in *authData* are as expected, considering the client extension input values that were given in `options.extensions` and any specific policy of the Relying Party regarding unsolicited extensions, i.e., those that were not specified as part of `options.extensions`. In the general case, the meaning of "are as expected" is specific to the Relying Party and which extensions are in use.

> Note: Client platforms MAY enact local policy that sets additional authenticator extensions or client extensions and thus cause values to appear in the authenticator extension outputs or client extension outputs that were not originally specified as part of `options.extensions`. Relying Parties MUST be prepared to handle such situations, whether it be to ignore the unsolicited extensions or reject the assertion. The Relying Party can make this decision based on local policy and the extensions in use.

> Note: Since all extensions are OPTIONAL for both the client and the authenticator, the Relying Party MUST also be prepared to handle cases where none or not all of the requested extensions were acted upon.

19. Let *hash* be the result of computing a hash over the *cData* using SHA-256.
20. Using *credentialPublicKey*, verify that *sig* is a valid signature over the binary concatenation of *authData* and *hash*.

> Note: This verification step is compatible with signatures generated by FIDO U2F authenticators. See § 6.1.2 FIDO U2F Signature Format Compatibility.

21. Let *storedSignCount* be the stored signature counter value associated with `credential.id`. If *authData*.`signCount` is nonzero or *storedSignCount* is nonzero, then run the following sub-step:

    - If *authData*.`signCount` is

      ↪ **greater than *storedSignCount*:**
      Update *storedSignCount* to be the value of *authData*.`signCount`.

      ↪ **less than or equal to *storedSignCount*:**
      This is a signal that the authenticator may be cloned, i.e. at least two copies of the credential private key may exist and are being used in parallel. Relying Parties should incorporate this information into their risk scoring. Whether the Relying Party updates *storedSignCount* in this case, or not, or fails the authentication ceremony or not, is Relying Party-specific.

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

§ 8. Defined Attestation Statement Formats

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

§ 8.1. Attestation Statement Format Identifiers

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

Attestation statement format identifiers SHOULD be registered in the IANA "WebAuthn Attestation Statement Format Identifiers" registry [IANA-WebAuthn-Registries] established by [RFC8809]. All registered attestation statement format identifiers are unique amongst themselves as a matter of course.

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

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

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

Attestation statement formats that may exist in multiple versions SHOULD include a version in their identifier. In effect, different versions are thus treated as different formats, e.g., `packed2` as a new version of the § 8.2 Packed Attestation Statement Format.

The following sections present a set of currently-defined and registered attestation statement formats and their identifiers. The up-to-date list of registered WebAuthn Extensions is maintained in the IANA "WebAuthn Attestation Statement Format Identifiers" registry [IANA-WebAuthn-Registries] established by [RFC8809].

§ 8.2. Packed Attestation Statement Format

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

Attestation statement format identifier

packed

Attestation types supported

Basic, Self, AttCA

Syntax

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

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

packedStmtFormat = {
                       alg: COSEAlgorithmIdentifier,
                       sig: bytes,
                       x5c: [ attestnCert: bytes, * (caCert: bytes) ]
                     } //
                     {
                       alg: COSEAlgorithmIdentifier
                       sig: bytes,
                     }
```

The semantics of the fields are as follows:

alg

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

sig

A byte string containing the attestation signature.

x5c

The elements of this array contain *attestnCert* and its certificate chain (if any), each encoded in X.509 format. The attestation certificate *attestnCert* MUST be the first element in the array.

attestnCert

The attestation certificate, encoded in X.509 format.

Signing procedure

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

1. Let *authenticatorData* denote the authenticator data for the attestation, and let *clientDataHash* denote the hash of the serialized client data.
2. If Basic or AttCA attestation is in use, the authenticator produces the *sig* by concatenating *authenticatorData* and *clientDataHash*, and signing the result using an attestation private key selected through an authenticator-specific mechanism. It sets *x5c* to *attestnCert* followed by the related certificate chain (if any). It sets *alg* to the algorithm of the attestation private key.
3. If self attestation is in use, the authenticator produces *sig* by concatenating *authenticatorData* and *clientDataHash*, and signing the result using the credential private key. It sets *alg* to the algorithm of the credential private key and omits the other fields.

Verification procedure

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

1. Verify that *attStmt* is valid CBOR conforming to the syntax defined above and perform CBOR decoding on it to extract the contained fields.
2. If *x5c* is present:
    - Verify that *sig* is a valid signature over the concatenation of *authenticatorData* and *clientDataHash* using the attestation public key in *attestnCert* with the algorithm specified in *alg*.
    - Verify that *attestnCert* meets the requirements in § 8.2.1 Packed Attestation Statement Certificate Requirements.

- If *attestnCert* contains an extension with OID `1.3.6.1.4.1.45724.1.1.4` (`id-fido-gen-ce-aaguid`) verify that the value of this extension matches the `aaguid` in *authenticatorData*.
- Optionally, inspect *x5c* and consult externally provided knowledge to determine whether *attStmt* conveys a Basic or AttCA attestation.
- If successful, return implementation-specific values representing attestation type Basic, AttCA or uncertainty, and attestation trust path *x5c*.

3. If *x5c* is not present, self attestation is in use.
   - Validate that *alg* matches the algorithm of the `credentialPublicKey` in *authenticatorData*.
   - Verify that *sig* is a valid signature over the concatenation of *authenticatorData* and *clientDataHash* using the credential public key with *alg*.
   - If successful, return implementation-specific values representing attestation type Self and an empty attestation trust path.

§ 8.2.1. Packed Attestation Statement Certificate Requirements

The attestation certificate MUST have the following fields/extensions:

- Version MUST be set to 3 (which is indicated by an ASN.1 INTEGER with value 2).
- Subject field MUST be set to:

  Subject-C
  : ISO 3166 code specifying the country where the Authenticator vendor is incorporated (PrintableString)

  Subject-O
  : Legal name of the Authenticator vendor (UTF8String)

Subject-OU
: Literal string “Authenticator Attestation” (UTF8String)

Subject-CN
: A UTF8String of the vendor’s choosing

- 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 a 16-byte OCTET STRING. The extension MUST NOT be marked as critical.

  Note that an X.509 Extension encodes the DER-encoding of the value in an OCTET STRING. Thus, the AAGUID MUST be wrapped in *two* OCTET STRINGS to be valid. Here is a sample, encoded Extension structure:

```
30 21                                           -- SEQUENCE
  06 0b 2b 06 01 04 01 82 e5 1c 01 01 04        -- 1.3.6.1.4.1.45724.1.1.4
  04 12                                         -- OCTET STRING
    04 10                                       -- OCTET STRING
      cd 8c 39 5c 26 ed ee de                   -- AAGUID
      65 3b 00 79 7d 03 ca 3c
```

- 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].

§ 8.3. TPM Attestation Statement Format

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

Attestation statement format identifier

tpm

Attestation types supported

AttCA

Syntax

The syntax of a TPM Attestation statement is as follows:

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

tpmStmtFormat = {
                     ver: "2.0",
                     (
                         alg: COSEAlgorithmIdentifier,
                         x5c: [ aikCert: bytes, * (caCert: bytes) ]
                     )
                     sig: bytes,
                     certInfo: bytes,
                     pubArea: bytes
                 }
```

The semantics of the above fields are as follows:

ver

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

alg

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

x5c

aikCert followed by its certificate chain, in X.509 encoding.

aikCert

The AIK certificate used for the attestation, in X.509 encoding.

sig

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

certInfo

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

pubArea

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

Signing procedure

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

Concatenate *authenticatorData* and *clientDataHash* to form *attToBeSigned*.

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

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

Verification procedure

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

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

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

Concatenate *authenticatorData* and *clientDataHash* to form *attToBeSigned*.

Validate that *certInfo* is valid:

- Verify that `magic` is set to `TPM_GENERATED_VALUE`.
- Verify that `type` is set to `TPM_ST_ATTEST_CERTIFY`.
- Verify that `extraData` is set to the hash of *attToBeSigned* using the hash algorithm employed in "alg".
- Verify that `attested` contains a `TPMS_CERTIFY_INFO` structure as specified in [TPMv2-Part2] section 10.12.3, whose `name` field contains a valid Name for *pubArea*, as computed using the algorithm in the `nameAlg` field of *pubArea* using the procedure specified in [TPMv2-Part1] section 16.
- Verify that *x5c* is present.
- Note that the remaining fields in the "Standard Attestation Structure" [TPMv2-Part1] section 31.2, i.e., `qualifiedSigner`, `clockInfo` and `firmwareVersion` are ignored. These fields MAY be used as an input to risk engines.
- Verify the *sig* is a valid signature over *certInfo* using the attestation public key in *aikCert* with the algorithm specified in *alg*.
- Verify that *aikCert* meets the requirements in § 8.3.1 TPM Attestation Statement Certificate Requirements.
- If *aikCert* contains an extension with OID `1.3.6.1.4.1.45724.1.1.4` (`id-fido-gen-ce-aaguid`) verify that the value of this extension matches the `aaguid` in *authenticatorData*.

- If successful, return implementation-specific values representing attestation type AttCA and attestation trust path *x5c*.

8.3.1. TPM Attestation Statement Certificate Requirements

TPM attestation certificate MUST have the following fields/extensions:

- Version MUST be set to 3.
- Subject field MUST be set to empty.
- The Subject Alternative Name extension MUST be set as defined in [TPMv2-EK-Profile] section 3.2.9.
- The Extended Key Usage extension MUST contain the OID `2.23.133.8.3` ("joint-iso-itu-t(2) internationalorganizations(23) 133 tcg-kp(8) tcg-kp-AIKCertificate(3)").
- The Basic Constraints extension MUST have the CA component set to `false`.
- An Authority Information Access (AIA) extension with entry `id-ad-ocsp` and a CRL Distribution Point extension [RFC5280] are both OPTIONAL as the status of many attestation certificates is available through metadata services. See, for example, the FIDO Metadata Service [FIDOMetadataService].

8.4. Android Key Attestation Statement Format

When the authenticator in question is a platform 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 WebAuthn Relying Party is expected to check that the authenticator data claimed to have been used for the attestation is consistent with the fields of the attestation certificate's extension data.

Attestation statement format identifier

android-key

Attestation types supported

Basic

Syntax

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

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

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

Signing procedure

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

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

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

Verification procedure

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

- Verify that *attStmt* is valid CBOR conforming to the syntax defined above and perform CBOR decoding on it to extract the contained fields.
- Verify that *sig* is a valid signature over the concatenation of *authenticatorData* and *clientDataHash* using the public key in the first certificate in *x5c* with the algorithm specified in *alg*.
- Verify that the public key in the first certificate in *x5c* matches the `credentialPublicKey` in the `attestedCredentialData` in *authenticatorData*.
- Verify that the `attestationChallenge` field in the attestation certificate extension data is identical to *clientDataHash*.
- Verify the following using the appropriate authorization list from the attestation certificate extension data:
  - The `AuthorizationList.allApplications` field is *not* present on either authorization list (`softwareEnforced` nor `teeEnforced`), since PublicKeyCredential MUST be scoped to the RP ID.
  - For the following, use only the `teeEnforced` authorization list if the RP wants to accept only keys from a trusted execution environment, otherwise use the union of `teeEnforced` and `softwareEnforced`.
    - The value in the `AuthorizationList.origin` field is equal to `KM_ORIGIN_GENERATED`.
    - The value in the `AuthorizationList.purpose` field is equal to `KM_PURPOSE_SIGN`.
- If successful, return implementation-specific values representing attestation type Basic and attestation trust path *x5c*.

§ 8.4.1. Android Key Attestation Statement Certificate Requirements

Android Key Attestation attestation certificate's ***android key attestation certificate extension data*** is identified by the OID `1.3.6.1.4.1.11129.2.1.17`, and its schema is defined in the Android developer documentation.

§ 8.5. Android SafetyNet Attestation Statement Format

When the authenticator is a platform authenticator on certain Android platforms, the attestation statement may be 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 provides some statements about the health of the platform and the identity of the calling application (see SafetyNet Documentation for more details).

Attestation statement format identifier

android-safetynet

Attestation types supported

Basic

Syntax

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

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

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

The semantics of the above fields are as follows:

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

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

Signing procedure

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

Concatenate *authenticatorData* and *clientDataHash*, perform SHA-256 hash of the concatenated string, and let the result of the hash form *attToBeSigned*.

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

Verification procedure

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

- Verify that *attStmt* is valid CBOR conforming to the syntax defined above and perform CBOR decoding on it to extract the contained fields.
- Verify that *response* is a valid SafetyNet response of version *ver* by following the steps indicated by the SafetyNet online documentation. As of this writing, there is only one format of the SafetyNet response and *ver* is reserved for future use.
- Verify that the `nonce` attribute in the payload of *response* is identical to the Base64 encoding of the SHA-256 hash of the concatenation of *authenticatorData* and *clientDataHash*.

- Verify that the SafetyNet response actually came from the SafetyNet service by following the steps in the SafetyNet online documentation.
- If successful, return implementation-specific values representing attestation type Basic and attestation trust path *x5c*.

§ 8.6. FIDO U2F Attestation Statement Format

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

Attestation statement format identifier
fido-u2f

Attestation types supported
Basic, AttCA

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

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

u2fStmtFormat = {
                      x5c: [ attestnCert: bytes ],
                      sig: bytes
                  }
```

The semantics of the above fields are as follows:

x5c

A single element array containing the attestation certificate in X.509 format.

sig

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

Signing procedure

If the credential public key of the attested credential is not of algorithm -7 ("ES256"), stop and return an error. Otherwise, let *authenticatorData* denote the authenticator data for the attestation, and let *clientDataHash* denote the hash of the serialized client data. (Since SHA-256 is used to hash the serialized client data, *clientDataHash* will be 32 bytes long.)

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

Verification procedure

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

1. Verify that *attStmt* is valid CBOR conforming to the syntax defined above and perform CBOR decoding on it to extract the contained fields.
2. Check that *x5c* has exactly one element and let *attCert* be that element. Let *certificate public key* be the public key conveyed by *attCert*. If *certificate public key* is not an Elliptic Curve (EC) public key over the P-256 curve, terminate this algorithm and return an appropriate error.
3. Extract the claimed *rpIdHash* from *authenticatorData*, and the claimed *credentialId* and *credentialPublicKey* from *authenticatorData*.`attestedCredentialData`.

4. Convert the COSE_KEY formatted *credentialPublicKey* (see Section 7 of [RFC8152]) to Raw ANSI X9.62 public key format (see ALG_KEY_ECC_X962_RAW in Section 3.6.2 Public Key Representation Formats of [FIDO-Registry]).

    - Let *x* be the value corresponding to the "-2" key (representing x coordinate) in *credentialPublicKey*, and confirm its size to be of 32 bytes. If size differs or "-2" key is not found, terminate this algorithm and return an appropriate error.
    - Let *y* be the value corresponding to the "-3" key (representing y coordinate) in *credentialPublicKey*, and confirm its size to be of 32 bytes. If size differs or "-3" key is not found, terminate this algorithm and return an appropriate error.
    - Let *publicKeyU2F* be the concatenation `0x04 || x || y`.

        > Note: This signifies uncompressed ECC key format.

5. Let *verificationData* be the concatenation of (0x00 || *rpIdHash* || *clientDataHash* || *credentialId* || *publicKeyU2F*) (see Section 4.3 of [FIDO-U2F-Message-Formats]).
6. Verify the *sig* using *verificationData* and the *certificate public key* per section 4.1.4 of [SEC1] with SHA-256 as the hash function used in step two.
7. Optionally, inspect *x5c* and consult externally provided knowledge to determine whether *attStmt* conveys a Basic or AttCA attestation.
8. If successful, return implementation-specific values representing attestation type Basic, AttCA or uncertainty, and attestation trust path *x5c*.

§ 8.7. None Attestation Statement Format

The none attestation statement format is used to replace any authenticator-provided attestation statement when a WebAuthn Relying Party indicates it does not wish to receive attestation information, see § 5.4.7 Attestation Conveyance Preference Enumeration (enum AttestationConveyancePreference).

The authenticator MAY also directly generate attestation statements of this format if the authenticator does not support attestation.

Attestation statement format identifier

none

Attestation types supported

None

Syntax

The syntax of a none attestation statement is defined as follows:

```
$$attStmtType //= (
                        fmt: "none",
                        attStmt: emptyMap
                    )

emptyMap = {}
```

Signing procedure

Return the fixed attestation statement defined above.

Verification procedure

Return implementation-specific values representing attestation type None and an empty attestation trust path.

§ 8.8. Apple Anonymous Attestation Statement Format

This attestation statement format is exclusively used by Apple for certain types of Apple devices that support WebAuthn.

Attestation statement format identifier

apple

Attestation types supported

Anonymization CA

Syntax

The syntax of an Apple attestation statement is defined as follows:

```
$$attStmtType //= (
                        fmt: "apple",
                        attStmt: appleStmtFormat
                    )

appleStmtFormat = {
                        x5c: [ credCert: bytes, * (caCert: bytes) ]
                    }
```

The semantics of the above fields are as follows:

x5c

credCert followed by its certificate chain, each encoded in X.509 format.

credCert

The credential public key certificate used for attestation, encoded in X.509 format.

Signing procedure

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

2. Concatenate *authenticatorData* and *clientDataHash* to form *nonceToHash*.

3. Perform SHA-256 hash of *nonceToHash* to produce *nonce*.
4. Let Apple anonymous attestation CA generate an X.509 certificate for the credential public key and include the *nonce* as a certificate extension with OID `1.2.840.113635.100.8.2`. *credCert* denotes this certificate. The *credCert* thus serves as a proof of the attestation, and the included *nonce* proves the attestation is live. In addition to that, the *nonce* also protects the integrity of the *authenticatorData* and client data.
5. Set *x5c* to *credCert* followed by its certificate chain.

Verification procedure

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

1. Verify that *attStmt* is valid CBOR conforming to the syntax defined above and perform CBOR decoding on it to extract the contained fields.
2. Concatenate *authenticatorData* and *clientDataHash* to form *nonceToHash*.
3. Perform SHA-256 hash of *nonceToHash* to produce *nonce*.
4. Verify that *nonce* equals the value of the extension with OID `1.2.840.113635.100.8.2` in *credCert*.
5. Verify that the credential public key equals the Subject Public Key of *credCert*.
6. If successful, return implementation-specific values representing attestation type Anonymization CA and attestation trust path *x5c*.

§ 9. *WebAuthn Extensions*

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

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

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

When creating a public key credential or requesting an authentication assertion, a WebAuthn 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 WebAuthn Authenticator. The Relying Party sends the client extension input for each extension in the `get()` call (for authentication extensions) or `create()` call (for registration extensions) to the client. The client performs client extension processing for each extension that the client platform supports, and augments the client data as specified by each extension, by including the extension identifier and client extension output values.

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

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

For authenticator extensions, as part of the client extension processing, the client also creates the CBOR authenticator extension input value for each extension (often based on the corresponding client extension input value), and passes them to the authenticator in the `create()` call (for registration extensions) or the `get()` call (for authentication extensions). These authenticator extension input values are represented in CBOR and passed as name-value pairs, with the extension identifier as the name, and the corresponding authenticator extension input as the value. The authenticator, in turn, performs

additional processing for the extensions that it supports, and returns the CBOR authenticator extension output for each as specified by the extension. Part of the client extension processing for authenticator extensions is to use the authenticator extension output as an input to creating the client extension output.

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

Clients wishing to support the widest possible range of extensions MAY choose to pass through any extensions that they do not recognize to authenticators, generating the authenticator extension input by simply encoding the client extension input in CBOR. All WebAuthn Extensions MUST be defined in such a way that this implementation choice does not endanger the user's security or privacy. For instance, if an extension requires client processing, it could be defined in a manner that ensures such a naïve 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.

When clients choose to pass through extensions they do not recognize, the JavaScript values in the client extension inputs are converted to CBOR values in the authenticator extension inputs. When the JavaScript value is an %ArrayBuffer%, it is converted to a CBOR byte array. When the JavaScript value is a non-integer number, it is converted to a 64-bit CBOR floating point number. Otherwise, when the JavaScript type corresponds to a JSON type, the conversion is done using the rules defined in Section 6.2 of [RFC8949] (Converting from JSON to CBOR), but operating on inputs of JavaScript type values rather than inputs of JSON type values. Once these conversions are done, canonicalization of the resulting CBOR MUST be performed using the CTAP2 canonical CBOR encoding form.

Note that the JavaScript numeric conversion rules have the consequence that when a client passes through an extension it does not recognize, if the extension uses floating point values, authenticators need to be prepared to receive those values as CBOR integers, should the authenticator want the extension to always work without actual client support for it. This will happen when the floating point values used happen to be integers.

Likewise, when clients receive outputs from extensions they have passed through that they do not recognize, the CBOR values in the authenticator extension outputs are converted to JavaScript values in the client extension outputs. When the CBOR value is a byte string, it is converted to a JavaScript %ArrayBuffer% (rather than a base64url-encoded string). Otherwise, when the CBOR type corresponds to a JSON type, the conversion is done using the rules defined in Section 6.1 of [RFC8949] (Converting from CBOR to JSON), but producing outputs of JavaScript type values rather than outputs of JSON type values.

Note that some clients may choose to implement this pass-through capability under a feature flag. Supporting this capability can facilitate innovation, allowing authenticators to experiment with new extensions and Relying Parties to use them before there is explicit support for them in clients.

The IANA "WebAuthn Extension Identifiers" registry [IANA-WebAuthn-Registries] established by [RFC8809] can be consulted for an up-to-date list of registered WebAuthn Extensions.

§ 9.1. Extension Identifiers

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

Extension identifiers SHOULD be registered in the IANA "WebAuthn Extension Identifiers" registry [IANA-WebAuthn-Registries] established by [RFC8809]. All registered extension identifiers are unique amongst themselves as a matter of course.

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

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

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

§ 10 Defined Extensions defines an additional set of extensions and their identifiers. See the IANA "WebAuthn Extension Identifiers" registry [IANA-WebAuthn-Registries] established by [RFC8809] for an up-to-date list of registered WebAuthn Extension Identifiers.

§ 9.2. Defining Extensions

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

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

§ 9.3. Extending Request Parameters

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

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

> Note: Other documents have specified extensions where the extension input does not always use the extension identifier as the entry key. New extensions SHOULD follow the above convention.

```
EXAMPLE 10
var assertionPromise = navigator.credentials.get({
    publicKey: {
        // Other members omitted for brevity
        extensions: {
            // An "entry key" identifying the "webauthnExample_foobar" extension,
            // whose value is a map with two input parameters:
            "webauthnExample_foobar": {
              foo: 42,
              bar: "barfoo"
            }
        }
    }
});
```

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

Extensions that only affect client processing need not specify authenticator extension input. Extensions that have authenticator processing MUST specify the method of computing the authenticator extension input from the client extension input, and MUST define extensions for the CDDL types `AuthenticationExtensionsAuthenticatorInputs` and `AuthenticationExtensionsAuthenticatorOutputs` by defining an additional choice for the `$$extensionInput` and `$$extensionOutput` group sockets using the extension identifier as the entry key. Extensions that do not require input parameters, and are thus defined as taking a Boolean client extension input value set to `true`, SHOULD define the authenticator extension input also as the constant Boolean value `true` (CBOR major type 7, value 21).

The following example defines that an extension with identifier `webauthnExample_foobar` takes an unsigned integer as authenticator extension input, and returns an array of at least one byte string as authenticator extension output:

```
EXAMPLE 11
$$extensionInput //= (
  webauthnExample_foobar: uint
)
$$extensionOutput //= (
  webauthnExample_foobar: [+ bytes]
)
```

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

§ 9.4. *Client Extension Processing*

Extensions MAY define additional processing requirements on the client during the creation of credentials or the generation of an assertion. The client extension input for the extension is used as an input to this client processing. For each supported client extension, the client adds an entry to the *clientExtensions* map with the extension identifier as the key, and the extension's client extension input as the value.

Likewise, the client extension outputs are represented as a dictionary in the result of `getClientExtensionResults()` with extension identifiers as keys, and the ***client extension output*** value of each extension as the value. Like the client extension input, the client extension output is a value that can be encoded in JSON. There MUST NOT be any values returned for ignored extensions.

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

§ 9.5. *Authenticator Extension Processing*

The CBOR authenticator extension input value of each processed authenticator extension is included in the *extensions* parameter of the authenticatorMakeCredential and authenticatorGetAssertion operations. The *extensions* parameter is a CBOR map where each key is an extension identifier and the corresponding value is the authenticator extension input for that extension.

Likewise, the extension output is represented in the extensions part of the authenticator data. The extensions part of the authenticator data is a CBOR map where each key is an extension identifier and the corresponding value is the ***authenticator extension output*** for that extension.

For each supported extension, the authenticator extension processing rule for that extension is used create the authenticator extension output from the authenticator extension input and possibly also other inputs. There MUST NOT be any values

returned for ignored extensions.

§ 10. Defined Extensions

This section defines an additional set of extensions to be registered in the IANA "WebAuthn Extension Identifiers" registry [IANA-WebAuthn-Registries] established by [RFC8809]. These MAY be implemented by user agents targeting broad interoperability.

§ 10.1. FIDO *AppID* Extension (appid)

This extension allows WebAuthn Relying Parties that have previously registered a credential using the legacy FIDO U2F JavaScript API [FIDOU2FJavaScriptAPI] to request an assertion. The FIDO APIs use an alternative identifier for Relying Parties called an *AppID* [FIDO-APPID], and any credentials created using those APIs will be scoped to that identifier. Without this extension, they would need to be re-registered in order to be scoped to an RP ID.

In addition to setting the `appid` extension input, using this extension requires some additional processing by the Relying Party in order to allow users to authenticate using their registered U2F credentials:

1. List the desired U2F credentials in the `allowCredentials` option of the `get()` method:
    - Set the `type` members to `public-key`.
    - Set the `id` members to the respective U2F key handles of the desired credentials. Note that U2F key handles commonly use base64url encoding but must be decoded to their binary form when used in `id`.

    `allowCredentials` MAY contain a mixture of both WebAuthn credential IDs and U2F key handles; stating the `appid` via this extension does not prevent the user from using a WebAuthn-registered credential scoped to the RP ID stated in `rpId`.

2. When verifying the assertion, expect that the `rpIdHash` MAY be the hash of the *AppID* instead of the RP ID.

This extension does not allow FIDO-compatible credentials to be created. Thus, credentials created with WebAuthn are not backwards compatible with the FIDO JavaScript APIs.

> Note: `appid` should be set to the AppID that the Relying Party *previously* used in the legacy FIDO APIs. This might not be the same as the result of translating the Relying Party's WebAuthn RP ID to the AppID format, e.g., the previously used AppID may have been "https://accounts.example.com" but the currently used RP ID might be "example.com".

Extension identifier

`appid`

Operation applicability

Authentication

Client extension input

A single USVString specifying a FIDO *AppID*.

```
partial dictionary AuthenticationExtensionsClientInputs {
  USVString appid;
};
```

Client extension processing

1. Let *facetId* be the result of passing the caller's origin to the FIDO algorithm for determining the FacetID of a calling application.
2. Let *appId* be the extension input.
3. Pass *facetId* and *appId* to the FIDO algorithm for determining if a caller's FacetID is authorized for an AppID. If that algorithm rejects *appId* then return a "`SecurityError`" `DOMException`.

4. When building allowCredentialDescriptorList, if a U2F authenticator indicates that a credential is inapplicable (i.e. by returning `SW_WRONG_DATA`) then the client MUST retry with the U2F application parameter set to the SHA-256 hash of *appId*. If this results in an applicable credential, the client MUST include the credential in *allowCredentialDescriptorList*. The value of *appId* then replaces the `rpId` parameter of authenticatorGetAssertion.

5. Let *output* be the Boolean value `false`.

6. When creating assertionCreationData, if the assertion was created by a U2F authenticator with the U2F application parameter set to the SHA-256 hash of *appId* instead of the SHA-256 hash of the RP ID, set *output* to `true`.

> Note: In practice, several implementations do not implement steps four and onward of the algorithm for determining if a caller's FacetID is authorized for an AppID. Instead, in step three, the comparison on the host is relaxed to accept hosts on the same site.

Client extension output

Returns the value of *output*. If true, the *AppID* was used and thus, when verifying the assertion, the Relying Party MUST expect the `rpIdHash` to be the hash of the *AppID*, not the RP ID.

```
partial dictionary AuthenticationExtensionsClientOutputs {
  boolean appid;
};
```

Authenticator extension input

None.

Authenticator extension processing

None.

Authenticator extension output

None.

§ 10.2. FIDO AppID Exclusion Extension (appidExclude)

This registration extension allows WebAuthn Relying Parties to exclude authenticators that contain specified credentials that were created with the legacy FIDO U2F JavaScript API [FIDOU2FJavaScriptAPI].

During a transition from the FIDO U2F JavaScript API, a Relying Party may have a population of users with legacy credentials already registered. The appid extension allows the sign-in flow to be transitioned smoothly but, when transitioning the registration flow, the excludeCredentials field will not be effective in excluding authenticators with legacy credentials because its contents are taken to be WebAuthn credentials. This extension directs client platforms to consider the contents of excludeCredentials as both WebAuthn and legacy FIDO credentials. Note that U2F key handles commonly use base64url encoding but must be decoded to their binary form when used in excludeCredentials.

Extension identifier

`appidExclude`

Operation applicability

Registration

Client extension input

A single USVString specifying a FIDO *AppID*.

```
partial dictionary AuthenticationExtensionsClientInputs {
  USVString appidExclude;
};
```

Client extension processing

When creating a new credential:

1. Just after establishing the RP ID perform these steps:
    1. Let *facetId* be the result of passing the caller's origin to the FIDO algorithm for determining the FacetID of a calling application.

2. Let *appId* be the value of the extension input `appidExclude`.
3. Pass *facetId* and *appId* to the FIDO algorithm for determining if a caller's FacetID is authorized for an AppID. If the latter algorithm rejects *appId* then return a "`SecurityError`" `DOMException` and terminate the creating a new credential algorithm as well as these steps.

> Note: In practice, several implementations do not implement steps four and onward of the algorithm for determining if a caller's FacetID is authorized for an AppID. Instead, in step three, the comparison on the host is relaxed to accept hosts on the same site.

4. Otherwise, continue with normal processing.

2. Just prior to invoking authenticatorMakeCredential perform these steps:

1. If *authenticator* supports the U2F protocol [FIDO-U2F-Message-Formats], then for each credential descriptor *C* in *excludeCredentialDescriptorList*:

1. Check whether *C* was created using U2F on *authenticator* by sending a `U2F_AUTHENTICATE` message to *authenticator* whose "five parts" are set to the following values:

control byte
`0x07` ("check-only")

challenge parameter
32 random bytes

application parameter
SHA-256 hash of *appId*

key handle length
The length of `C.id` (in bytes)

key handle
The value of `C.id`, i.e., the credential id.

2. If *authenticator* responds with `message:error:test-of-user-presence-required` (i.e., success): cease normal processing of this *authenticator* and indicate in a platform-specific manner that the authenticator is inapplicable. For example, this could be in the form of UI, or could involve requesting user consent from *authenticator* and, upon receipt, treating it as if the authenticator had returned `InvalidStateError`. Requesting user consent can be accomplished by sending another `U2F_AUTHENTICATE` message to *authenticator* as above except for setting *control byte* to `0x03` ("enforce-user-presence-and-sign"), and ignoring the response.

2. Continue with normal processing.

Client extension output

Returns the value `true` to indicate to the Relying Party that the extension was acted upon.

```
partial dictionary AuthenticationExtensionsClientOutputs {
  boolean appidExclude;
};
```

Authenticator extension input

None.

Authenticator extension processing

None.

Authenticator extension output

None.

§ 10.3. *User Verification Method* Extension (uvm)

This extension enables use of a user verification method.

Extension identifier

`uvm`

Operation applicability

Registration and Authentication

Client extension input

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

```
partial dictionary AuthenticationExtensionsClientInputs {
  boolean uvm;
};
```

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.

```
typedef sequence<unsigned long> UvmEntry;
typedef sequence<UvmEntry> UvmEntries;

partial dictionary AuthenticationExtensionsClientOutputs {
  UvmEntries uvm;
};
```

Authenticator extension input

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

```
$$extensionInput //= (
  uvm: true,
)
```

Authenticator extension processing

The authenticator sets the authenticator extension output to be one or more user verification methods indicating the method(s) 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:

```
$$extensionOutput //= (
  uvm: [ 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 Section 3.1 User Verification Methods of [FIDO-Registry].

keyProtectionType

The method used by the authenticator to protect the FIDO registration private key material. Available values are defined in Section 3.2 Key Protection Types of [FIDO-Registry].

matcherProtectionType

The method used by the authenticator to protect the matcher that performs user verification. Available values are defined in Section 3.3 Matcher Protection Types of [FIDO-Registry].

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

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

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

§ 10.4. Credential Properties Extension (*credProps*)

This client registration extension facilitates reporting certain credential properties known by the client to the requesting WebAuthn Relying Party upon creation of a public key credential source as a result of a registration ceremony.

At this time, one credential property is defined: the resident key credential property (i.e., client-side discoverable credential property).

Extension identifier

`credProps`

Operation applicability

Registration

Client extension input

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

```
partial dictionary AuthenticationExtensionsClientInputs {
    boolean credProps;
};
```

Client extension processing

None, other than to report on credential properties in the output.

Client extension output

Set `clientExtensionResults["credProps"]["rk"]` to the value of the *requireResidentKey* parameter that was used in the invocation of the authenticatorMakeCredential operation.

```
dictionary CredentialPropertiesOutput {
    boolean rk;
};

partial dictionary AuthenticationExtensionsClientOutputs {
    CredentialPropertiesOutput credProps;
};
```

***rk*, of type boolean**

This OPTIONAL property, known abstractly as the ***resident key credential property*** (i.e., ***client-side discoverable credential property***), is a Boolean value indicating whether the `PublicKeyCredential` returned as a result of a registration ceremony is a client-side discoverable credential. If `rk` is `true`, the credential is a discoverable

credential. if `rk` is `false`, the credential is a server-side credential. If `rk` is not present, it is not known whether the credential is a discoverable credential or a server-side credential.

> Note: some authenticators create discoverable credentials even when not requested by the client platform. Because of this, client platforms may be forced to omit the `rk` property because they lack the assurance to be able to set it to `false`. Relying Parties should assume that, if the `credProps` extension is supported, then client platforms will endeavour to populate the `rk` property. Therefore a missing `rk` indicates that the created credential is most likely a non-discoverable credential.

Authenticator extension input

None.

Authenticator extension processing

None.

Authenticator extension output

None.

§ 10.5. Large blob storage extension (***largeBlob***)

This client registration extension and authentication extension allows a Relying Party to store opaque data associated with a credential. Since authenticators can only store small amounts of data, and most Relying Parties are online services that can store arbitrary amounts of state for a user, this is only useful in specific cases. For example, the Relying Party might wish to issue certificates rather than run a centralised authentication service.

> Note: Relying Parties can assume that the opaque data will be compressed when being written to a space-limited device and so need not compress it themselves.

Since a certificate system needs to sign over the public key of the credential, and that public key is only available after creation, this extension does not add an ability to write blobs in the registration context. However, Relying Parties SHOULD use the registration extension when creating the credential if they wish to later use the authentication extension.

Since certificates are sizable relative to the storage capabilities of typical authenticators, user agents SHOULD consider what indications and confirmations are suitable to best guide the user in allocating this limited resource and prevent abuse.

> Note: In order to interoperate, user agents storing large blobs on authenticators using [FIDO-CTAP] are expected to use the provisions detailed in that specification for storing large, per-credential blobs.

Extension identifier

`largeBlob`

Operation applicability

Registration and authentication

Client extension input

```
partial dictionary AuthenticationExtensionsClientInputs {
    AuthenticationExtensionsLargeBlobInputs largeBlob;
};

enum LargeBlobSupport {
  "required",
  "preferred",
};

dictionary AuthenticationExtensionsLargeBlobInputs {
    DOMString support;
```

```
    boolean read;
    BufferSource write;
};
```

***support*, of type DOMString**

A DOMString that takes one of the values of `LargeBlobSupport`. (See § 2.1.1 Enumerations as DOMString types.) Only valid during registration.

***read*, of type boolean**

A boolean that indicates that the Relying Party would like to fetch the previously-written blob associated with the asserted credential. Only valid during authentication.

***write*, of type BufferSource**

An opaque byte string that the Relying Party wishes to store with the existing credential. Only valid during authentication.

Client extension processing (registration)

1. If `read` or `write` is present:
    1. Return a `DOMException` whose name is "`NotSupportedError`".
2. If `support` is present and has the value `required`:
    1. Set `supported` to `true`.

    > Note: This is in anticipation of an authenticator capable of storing large blobs becoming available. It occurs during extension processing in Step 11 of `[[Create]]()`. The `AuthenticationExtensionsLargeBlobOutputs` will be abandoned if no satisfactory authenticator becomes available.

    2. If a candidate authenticator becomes available (Step 19 of `[[Create]]()`) then, before evaluating any *`options`*, continue (i.e. ignore the candidate authenticator) if the candidate authenticator is not capable of

storing large blobs.

3. Otherwise (i.e. `support` is absent or has the value `preferred`):

    1. If an authenticator is selected and the selected authenticator supports large blobs, set `supported` to `true`, and `false` otherwise.

Client extension processing (authentication)

1. If `support` is present:

    1. Return a `DOMException` whose name is "`NotSupportedError`".

2. If both `read` and `write` are present:

    1. Return a `DOMException` whose name is "`NotSupportedError`".

3. If `read` is present and has the value `true`:

    1. Initialize the client extension output, `largeBlob`.

    2. If any authenticator indicates success (in `[[DiscoverFromExternalSource]]()`), attempt to read any largeBlob data associated with the asserted credential.

    3. If successful, set `blob` to the result.

    > Note: if the read is not successful, `largeBlob` will be present in `AuthenticationExtensionsClientOutputs` but the `blob` member will not be present.

4. If `write` is present:

    1. If `allowCredentials` does not contain exactly one element:

        1. Return a `DOMException` whose name is "`NotSupportedError`".

2. If the assertion operation is successful, attempt to store the contents of `write` on the authenticator, associated with the indicated credential.

3. Set `written` to `true` if successful and `false` otherwise.

Client extension output

```
partial dictionary AuthenticationExtensionsClientOutputs {
    AuthenticationExtensionsLargeBlobOutputs largeBlob;
};

dictionary AuthenticationExtensionsLargeBlobOutputs {
    boolean supported;
    ArrayBuffer blob;
    boolean written;
};
```

supported, **of type boolean**

`true` if, and only if, the created credential supports storing large blobs. Only present in registration outputs.

blob, **of type ArrayBuffer**

The opaque byte string that was associated with the credential identified by `rawId`. Only valid if `read` was `true`.

written, **of type boolean**

A boolean that indicates that the contents of `write` were successfully stored on the authenticator, associated with the specified credential.

Authenticator extension processing

This extension directs the user-agent to cause the large blob to be stored on, or retrieved from, the authenticator. It thus does not specify any direct authenticator interaction for Relying Parties.

§ 11. User Agent Automation

For the purposes of user agent automation and web application testing, this document defines a number of [WebDriver] extension commands.

§ 11.1. WebAuthn WebDriver Extension Capability

In order to advertise the availability of the extension commands defined below, a new extension capability is defined.

Capability	Key	Value Type	Description
Virtual Authenticators Support	`"webauthn:virtualAuthenticators"`	boolean	Indicates whether the endpoint node supports all Virtual Authenticators commands.

When validating capabilities, the extension-specific substeps to validate `"webauthn:virtualAuthenticators"` with `value` are the following:

1. If `value` is not a boolean return a WebDriver Error with WebDriver error code invalid argument.
2. Otherwise, let `deserialized` be set to `value`.

When matching capabilities, the extension-specific steps to match `"webauthn:virtualAuthenticators"` with `value` are the following:

1. If `value` is `true` and the endpoint node does not support any of the Virtual Authenticators commands, the match is unsuccessful.
2. Otherwise, the match is successful.

11.1.1. *Authenticator Extension Capabilities*

Additionally, extension capabilities are defined for every authenticator extension (i.e. those defining authenticator extension processing) defined in this specification:

Capability	Key	Value Type	Description
User Verification Method Extension Support	`"webauthn:extension:uvm"`	boolean	Indicates whether the endpoint node WebAuthn WebDriver implementation supports the User Verification Method extension.
Large Blob Storage Extension Support	`"webauthn:extension:largeBlob"`	boolean	Indicates whether the endpoint node WebAuthn WebDriver implementation supports the largeBlob extension.

When validating capabilities, the extension-specific substeps to validate an authenticator extension capability `key` with `value` are the following:

1. If `value` is not a boolean return a WebDriver Error with WebDriver error code invalid argument.
2. Otherwise, let `deserialized` be set to `value`.

When matching capabilities, the extension-specific steps to match an authenticator extension capability `key` with `value` are the following:

1. If `value` is `true` and the endpoint node WebAuthn WebDriver implementation does not support the authenticator extension identified by the `key`, the match is unsuccessful.
2. Otherwise, the match is successful.

User-Agents implementing defined authenticator extensions SHOULD implement the corresponding authenticator extension capability.

§ 11.2. *Virtual Authenticators*

These WebDriver extension commands create and interact with Virtual Authenticators: software implementations of the Authenticator Model. Virtual Authenticators are stored in a ***Virtual Authenticator Database***. Each stored virtual authenticator has the following properties:

authenticatorId
: An non-null string made using up to 48 characters from the `unreserved` production defined in Appendix A of [RFC3986] that uniquely identifies the Virtual Authenticator.

protocol
: The protocol the Virtual Authenticator speaks: one of `"ctap1/u2f"`, `"ctap2"` or `"ctap2_1"` [FIDO-CTAP].

transport
: The `AuthenticatorTransport` simulated. If the *transport* is set to `internal`, the authenticator simulates platform attachment. Otherwise, it simulates cross-platform attachment.

hasResidentKey

If set to `true` the authenticator will support client-side discoverable credentials.

hasUserVerification

If set to `true`, the authenticator supports user verification.

isUserConsenting

Determines the result of all user consent authorization gestures, and by extension, any test of user presence performed on the Virtual Authenticator. If set to `true`, a user consent will always be granted. If set to `false`, it will not be granted.

isUserVerified

Determines the result of User Verification performed on the Virtual Authenticator. If set to `true`, User Verification will always succeed. If set to `false`, it will fail.

> Note: This property has no effect if *hasUserVerification* is set to `false`.

extensions

A string array containing the extension identifiers supported by the Virtual Authenticator.

A Virtual authenticator MUST support all authenticator extensions present in its *extensions* array. It MUST NOT support any authenticator extension not present in its *extensions* array.

uvm

A `UvmEntries` array to be set as the authenticator extension output when processing the User Verification Method extension.

> Note: This property has no effect if the Virtual Authenticator does not support the User Verification Method extension.

§ 11.3. *Add Virtual Authenticator*

The Add Virtual Authenticator WebDriver extension command creates a software Virtual Authenticator. It is defined as follows:

HTTP Method	URI Template
POST	`/session/{session id}/webauthn/authenticator`

The ***Authenticator Configuration*** is a JSON Object passed to the remote end steps as *parameters*. It contains the following *key* and *value* pairs:

Key	Value Type	Valid Values	Default
protocol	string	`"ctap1/u2f"`, `"ctap2"`, `"ctap2_1"`	None
transport	string	`AuthenticatorTransport` values	None
hasResidentKey	boolean	`true`, `false`	`false`
hasUserVerification	boolean	`true`, `false`	`false`
isUserConsenting	boolean	`true`, `false`	`true`
isUserVerified	boolean	`true`, `false`	`false`
extensions	string array	An array containing extension identifiers	Empty array
uvm	`UvmEntries`	Up to 3 User Verification Method entries	Empty array

The remote end steps are:

1. If *parameters* is not a JSON Object, return a WebDriver error with WebDriver error code invalid argument.

   > Note: *parameters* is a Authenticator Configuration object.

2. Let *authenticator* be a new Virtual Authenticator.
3. For each enumerable own property in *parameters*:

   1. Let *key* be the name of the property.

2. Let *value* be the result of getting a property named *key* from *parameters*.
3. If there is no matching `key` for *key* in *parameters*, return a WebDriver error with WebDriver error code invalid argument.
4. If *value* is not one of the `valid values` for that *key*, return a WebDriver error with WebDriver error code invalid argument.
5. Set a property *key* to *value* on *authenticator*.

4. For each property in Authenticator Configuration with a default defined:
    1. If `key` is not a defined property of *authenticator*, set a property `key` to `default` on *authenticator*.
5. For each property in Authenticator Configuration:
    1. If `key` is not a defined property of *authenticator*, return a WebDriver error with WebDriver error code invalid argument.
6. For each *extension* in *authenticator.extensions*:
    1. If *extension* is not an extension identifier supported by the endpoint node WebAuthn WebDriver implementation, return a WebDriver error with WebDriver error code unsupported operation.
7. Generate a valid unique authenticatorId.
8. Set a property `authenticatorId` to *authenticatorId* on *authenticator*.
9. Store *authenticator* in the Virtual Authenticator Database.
10. Return success with data *authenticatorId*.

§ 11.4. *Remove Virtual Authenticator*

The Remove Virtual Authenticator WebDriver extension command removes a previously created Virtual Authenticator. It is defined as follows:

HTTP Method	URI Template
DELETE	`/session/{session id}/webauthn/authenticator/{authenticatorId}`

The remote end steps are:

1. If *authenticatorId* does not match any Virtual Authenticator stored in the Virtual Authenticator Database, return a WebDriver error with WebDriver error code invalid argument.
2. Remove the Virtual Authenticator identified by *authenticatorId* from the Virtual Authenticator Database
3. Return success.

§ 11.5. *Add Credential*

The Add Credential WebDriver extension command injects a Public Key Credential Source into an existing Virtual Authenticator. It is defined as follows:

HTTP Method	URI Template
POST	`/session/{session id}/webauthn/authenticator/{authenticatorId}/credential`

The ***Credential Parameters*** is a JSON Object passed to the remote end steps as *parameters*. It contains the following *key* and *value* pairs:

Key	Description	Value Type
credentialId	The Credential ID encoded using Base64url Encoding.	string
isResidentCredential	If set to `true`, a client-side discoverable credential is created. If set to `false`, a server-side credential is created instead.	boolean
rpId	The Relying Party ID the credential is scoped to.	string
privateKey	An asymmetric key package containing a single private key per [RFC5958], encoded using Base64url Encoding.	string
userHandle	The userHandle associated to the credential encoded using Base64url Encoding. This property may not be defined.	string
signCount	The initial value for a signature counter associated to the public key credential source.	number
largeBlob	The large, per-credential blob associated to the public key credential source, encoded using Base64url Encoding. This property may not be defined.	string

The remote end steps are:

1. If *parameters* is not a JSON Object, return a WebDriver error with WebDriver error code invalid argument.

> Note: *parameters* is a Credential Parameters object.

2. Let *credentialId* be the result of decoding Base64url Encoding on the *parameters*' *credentialId* property.
3. If *credentialId* is failure, return a WebDriver error with WebDriver error code invalid argument.
4. Let *isResidentCredential* be the *parameters*' *isResidentCredential* property.
5. If *isResidentCredential* is not defined, return a WebDriver error with WebDriver error code invalid argument.
6. Let *rpId* be the *parameters*' *rpId* property.
7. If *rpId* is not a valid RP ID, return a WebDriver error with WebDriver error code invalid argument.
8. Let *privateKey* be the result of decoding Base64url Encoding on the *parameters*' *privateKey* property.
9. If *privateKey* is failure, return a WebDriver error with WebDriver error code invalid argument.
10. If *privateKey* is not a validly-encoded asymmetric key package containing a single ECDSA private key on the P-256 curve per [RFC5958], return a WebDriver error with WebDriver error code invalid argument.
11. If the *parameters*' *userHandle* property is defined:
    1. Let *userHandle* be the result of decoding Base64url Encoding on the *parameters*' *userHandle* property.
    2. If *userHandle* is failure, return a WebDriver error with WebDriver error code invalid argument.
12. Otherwise:
    1. If *isResidentCredential* is `true`, return a WebDriver error with WebDriver error code invalid argument.
    2. Let *userHandle* be `null`.
13. If *authenticatorId* does not match any Virtual Authenticator stored in the Virtual Authenticator Database, return a WebDriver error with WebDriver error code invalid argument.
14. Let *authenticator* be the Virtual Authenticator matched by *authenticatorId*.
15. If *isResidentCredential* is `true` and the *authenticator*'s *hasResidentKey* property is `false`, return a WebDriver error with WebDriver error code invalid argument.

16. If the *authenticator* supports the largeBlob extension and the *parameters*' *largeBlob* feature is defined:

    1. Let *largeBlob* be the result of decoding Base64url Encoding on the *parameters*' *largeBlob* property.
    2. If *largeBlob* is failure, return a WebDriver error with WebDriver error code invalid argument.

17. Otherwise:

    1. Let *largeBlob* be `null`.

18. Let *credential* be a new Client-side discoverable Public Key Credential Source if *isResidentCredential* is `true` or a Server-side Public Key Credential Source otherwise whose items are:

    type
    : `public-key`

    id
    : *credentialId*

    privateKey
    : *privateKey*

    rpId
    : *rpId*

    userHandle
    : *userHandle*

19. Associate a signature counter *counter* to the *credential* with a starting value equal to the *parameters*' *signCount* or `0` if *signCount* is `null`.

20. If *largeBlob* is not `null`, set the large, per-credential blob associated to the *credential* to *largeBlob*.

21. Store the *credential* and *counter* in the database of the *authenticator*.

22. Return success.

§ 11.6. *Get Credentials*

The Get Credentials WebDriver extension command returns one Credential Parameters object for every Public Key Credential Source stored in a Virtual Authenticator, regardless of whether they were stored using Add Credential or `navigator.credentials.create()`. It is defined as follows:

HTTP Method	URI Template
GET	`/session/{session id}/webauthn/authenticator/{authenticatorId}/credentials`

The remote end steps are:

1. If *authenticatorId* does not match any Virtual Authenticator stored in the Virtual Authenticator Database, return a WebDriver error with WebDriver error code invalid argument.
2. Let *credentialsArray* be an empty array.
3. For each Public Key Credential Source *credential*, managed by the authenticator identified by *authenticatorId*, construct a corresponding Credential Parameters Object and add it to *credentialsArray*.
4. Return success with data containing *credentialsArray*.

§ 11.7. *Remove Credential*

The Remove Credential WebDriver extension command removes a Public Key Credential Source stored on a Virtual Authenticator. It is defined as follows:

HTTP Method	URI Template
DELETE	`/session/{session id}/webauthn/authenticator/{authenticatorId}/credentials/{credentialId}`

The remote end steps are:

1. If *authenticatorId* does not match any Virtual Authenticator stored in the Virtual Authenticator Database, return a WebDriver error with WebDriver error code invalid argument.
2. Let *authenticator* be the Virtual Authenticator identified by *authenticatorId*.
3. If *credentialId* does not match any Public Key Credential Source managed by *authenticator*, return a WebDriver error with WebDriver error code invalid argument.
4. Remove the Public Key Credential Source identified by *credentialId* managed by *authenticator*.
5. Return success.

§ 11.8. *Remove All Credentials*

The Remove All Credentials WebDriver extension command removes all Public Key Credential Sources stored on a Virtual Authenticator. It is defined as follows:

HTTP Method	URI Template
DELETE	`/session/{session id}/webauthn/authenticator/{authenticatorId}/credentials`

The remote end steps are:

1. If *authenticatorId* does not match any Virtual Authenticator stored in the Virtual Authenticator Database, return a WebDriver error with WebDriver error code invalid argument.
2. Remove all Public Key Credential Sources managed by the Virtual Authenticator identified by *authenticatorId*.
3. Return success.

§ 11.9. *Set User Verified*

The Set User Verified extension command sets the *isUserVerified* property on the Virtual Authenticator. It is defined as follows:

HTTP Method	URI Template
POST	`/session/{session id}/webauthn/authenticator/{authenticatorId}/uv`

The remote end steps are:

1. If *parameters* is not a JSON Object, return a WebDriver error with WebDriver error code invalid argument.
2. If *authenticatorId* does not match any Virtual Authenticator stored in the Virtual Authenticator Database, return a WebDriver error with WebDriver error code invalid argument.
3. If *isUserVerified* is not a defined property of *parameters*, return a WebDriver error with WebDriver error code invalid argument.
4. Let *authenticator* be the Virtual Authenticator identified by *authenticatorId*.
5. Set the *authenticator*'s *isUserVerified* property to the *parameters*' *isUserVerified* property.
6. Return success.

§ 12. IANA Considerations

§ 12.1. WebAuthn Attestation Statement Format Identifier Registrations Updates

This section updates the below-listed attestation statement formats defined in Section § 8 Defined Attestation Statement Formats in the IANA "WebAuthn Attestation Statement Format Identifiers" registry [IANA-WebAuthn-Registries] established by [RFC8809], originally registered in [WebAuthn-1], to point to this specification.

- WebAuthn Attestation Statement Format Identifier: packed

- Description: The "packed" attestation statement format is a WebAuthn-optimized format for attestation. It uses a very compact but still extensible encoding method. This format is implementable by authenticators with limited resources (e.g., secure elements).
- Specification Document: Section § 8.2 Packed Attestation Statement Format of this specification

- WebAuthn Attestation Statement Format Identifier: tpm
- Description: The TPM attestation statement format returns an attestation statement in the same format as the packed attestation statement format, although the rawData and signature fields are computed differently.
- Specification Document: Section § 8.3 TPM Attestation Statement Format of this specification

- WebAuthn Attestation Statement Format Identifier: android-key
- Description: Platform authenticators on versions "N", and later, may provide this proprietary "hardware attestation" statement.
- Specification Document: Section § 8.4 Android Key Attestation Statement Format of this specification

- WebAuthn Attestation Statement Format Identifier: android-safetynet
- Description: Android-based platform authenticators MAY produce an attestation statement based on the Android SafetyNet API.
- Specification Document: Section § 8.5 Android SafetyNet Attestation Statement Format of this specification

- WebAuthn Attestation Statement Format Identifier: fido-u2f
- Description: Used with FIDO U2F authenticators
- Specification Document: Section § 8.6 FIDO U2F Attestation Statement Format of this specification

§ 12.2. WebAuthn Attestation Statement Format Identifier Registrations

This section registers the below-listed attestation statement formats, newly defined in Section § 8 Defined Attestation Statement Formats, in the IANA "WebAuthn Attestation Statement Format Identifiers" registry [IANA-WebAuthn-Registries] established by [RFC8809].

- WebAuthn Attestation Statement Format Identifier: apple
- Description: Used with Apple devices' platform authenticators
- Specification Document: Section § 8.8 Apple Anonymous Attestation Statement Format of this specification

- WebAuthn Attestation Statement Format Identifier: none
- Description: Used to replace any authenticator-provided attestation statement when a WebAuthn Relying Party indicates it does not wish to receive attestation information.
- Specification Document: Section § 8.7 None Attestation Statement Format of this specification

§ 12.3. WebAuthn Extension Identifier Registrations Updates

This section updates the below-listed extension identifier values defined in Section § 10 Defined Extensions in the IANA "WebAuthn Extension Identifiers" registry [IANA-WebAuthn-Registries] established by [RFC8809], originally registered in [WebAuthn-1], to point to this specification.

- WebAuthn Extension Identifier: appid
- Description: This authentication extension allows WebAuthn Relying Parties that have previously registered a credential using the legacy FIDO JavaScript APIs to request an assertion.
- Specification Document: Section § 10.1 FIDO AppID Extension (appid) of this specification

- WebAuthn Extension Identifier: uvm
- Description: This registration extension and authentication 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 § 10.3 User Verification Method Extension (uvm) of this specification

§ 12.4. WebAuthn Extension Identifier Registrations

This section registers the below-listed extension identifier values, newly defined in Section § 10 Defined Extensions, in the IANA "WebAuthn Extension Identifiers" registry [IANA-WebAuthn-Registries] established by [RFC8809].

- WebAuthn Extension Identifier: appidExclude
- Description: This registration extension allows WebAuthn Relying Parties to exclude authenticators that contain specified credentials that were created with the legacy FIDO U2F JavaScript API [FIDOU2FJavaScriptAPI].
- Specification Document: Section § 10.2 FIDO AppID Exclusion Extension (appidExclude) of this specification

- WebAuthn Extension Identifier: credProps
- Description: This client registration extension enables reporting of a newly-created credential's properties, as determined by the client, to the calling WebAuthn Relying Party's web application.
- Specification Document: Section § 10.4 Credential Properties Extension (credProps) of this specification

- WebAuthn Extension Identifier: largeBlob
- Description: This client registration extension and authentication extension allows a Relying Party to store opaque data associated with a credential.

- Specification Document: Section § 10.5 Large blob storage extension (largeBlob) of this specification

§ 13. Security Considerations

This specification defines a Web API and a cryptographic peer-entity authentication protocol. The Web Authentication API allows Web developers (i.e., "authors") to utilize the Web Authentication protocol in their registration and authentication ceremonies. The entities comprising the Web Authentication protocol endpoints are user-controlled WebAuthn Authenticators and a WebAuthn Relying Party's computing environment hosting the Relying Party's web application. In this model, the user agent, together with the WebAuthn Client, comprise an intermediary between authenticators and Relying Parties. Additionally, authenticators can attest to Relying Parties as to their provenance.

At this time, this specification does not feature detailed security considerations. However, the [FIDOSecRef] document provides a security analysis which is overall applicable to this specification. Also, the [FIDOAuthnrSecReqs] document suite provides useful information about authenticator security characteristics.

The below subsections comprise the current Web Authentication-specific security considerations. They are divided by audience; general security considerations are direct subsections of this section, while security considerations specifically for authenticator, client and Relying Party implementers are grouped into respective subsections.

§ 13.1. Credential ID Unsigned

The credential ID is not signed. This is not a problem because all that would happen if an authenticator returns the wrong credential ID, or if an attacker intercepts and manipulates the credential ID, is that the WebAuthn Relying Party would not look up the correct credential public key with which to verify the returned signed authenticator data (a.k.a., assertion), and thus the interaction would end in an error.

13.2. Physical Proximity between Client and Authenticator

In the WebAuthn authenticator model, it is generally assumed that roaming authenticators are physically close to, and communicate directly with, the client. This arrangement has some important advantages.

The promise of physical proximity between client and authenticator is a key strength of a something you have authentication factor. For example, if a roaming authenticator can communicate only via USB or Bluetooth, the limited range of these transports ensures that any malicious actor must physically be within that range in order to interact with the authenticator. This is not necessarily true of an authenticator that can be invoked remotely — even if the authenticator verifies user presence, users can be tricked into authorizing remotely initiated malicious requests.

Direct communication between client and authenticator means the client can enforce the scope restrictions for credentials. By contrast, if the communication between client and authenticator is mediated by some third party, then the client has to trust the third party to enforce the scope restrictions and control access to the authenticator. Failure to do either could result in a malicious Relying Party receiving authentication assertions valid for other Relying Parties, or in a malicious user gaining access to authentication assertions for other users.

If designing a solution where the authenticator does not need to be physically close to the client, or where client and authenticator do not communicate directly, designers SHOULD consider how this affects the enforcement of scope restrictions and the strength of the authenticator as a something you have authentication factor.

13.3. Security considerations for authenticators

13.3.1. 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 an authenticator model.

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.

§ 13.3.2. 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 key pairs are still safe although their certificates can no longer be trusted. A WebAuthn Authenticator manufacturer that has recorded the attestation public keys for their authenticator models 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 WebAuthn Relying Parties MUST update their trusted root certificates accordingly.

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

See also the related security consideration for Relying Parties in § 13.4.5 Revoked Attestation Certificates.

§ 13.4. Security considerations for Relying Parties

§ 13.4.1. Security Benefits for WebAuthn Relying Parties

The main benefits offered to WebAuthn Relying Parties by this specification include:

1. Users and accounts can be secured using widely compatible, easy-to-use multi-factor authentication.

2. The Relying Party does not need to provision authenticator hardware to its users. Instead, each user can independently obtain any conforming authenticator and use that same authenticator with any number of Relying Parties. The Relying Party can optionally enforce requirements on authenticators' security properties by inspecting the attestation statements returned from the authenticators.
3. Authentication ceremonies are resistant to man-in-the-middle attacks. Regarding registration ceremonies, see § 13.4.4 Attestation Limitations, below.
4. The Relying Party can automatically support multiple types of user verification - for example PIN, biometrics and/or future methods - with little or no code change, and can let each user decide which they prefer to use via their choice of authenticator.
5. The Relying Party does not need to store additional secrets in order to gain the above benefits.

As stated in the Conformance section, the Relying Party MUST behave as described in § 7 WebAuthn Relying Party Operations to obtain all of the above security benefits. However, one notable use case that departs slightly from this is described below in § 13.4.4 Attestation Limitations.

§ 13.4.2. Visibility Considerations for Embedded Usage

Simplistic use of WebAuthn in an embedded context, e.g., within `<iframe>`s as described in § 5.10 Using Web Authentication within iframe elements, may make users vulnerable to ***UI Redressing*** attacks, also known as "Clickjacking". This is where an attacker overlays their own UI on top of a Relying Party's intended UI and attempts to trick the user into performing unintended actions with the Relying Party. For example, using these techniques, an attacker might be able to trick users into purchasing items, transferring money, etc.

Even though WebAuthn-specific UI is typically handled by the client platform and thus is not vulnerable to UI Redressing, it is likely important for an Relying Party having embedded WebAuthn-wielding content to ensure that their content's UI is visible to the user. An emerging means to do so is by observing the status of the experimental Intersection Observer v2's

`isVisible` attribute. For example, the Relying Party's script running in the embedded context could pre-emptively load itself in a popup window if it detects `isVisble` being set to `false`, thus side-stepping any occlusion of their content.

§ 13.4.3. Cryptographic Challenges

As a cryptographic protocol, Web Authentication is dependent upon randomized challenges to avoid replay attacks. Therefore, the values of both `PublicKeyCredentialCreationOptions.challenge` and `PublicKeyCredentialRequestOptions.challenge` MUST be randomly generated by Relying Parties in an environment they trust (e.g., on the server-side), and the returned `challenge` value in the client's response MUST match what was generated. This SHOULD be done in a fashion that does not rely upon a client's behavior, e.g., the Relying Party SHOULD store the challenge temporarily until the operation is complete. Tolerating a mismatch will compromise the security of the protocol.

In order to prevent replay attacks, the challenges MUST contain enough entropy to make guessing them infeasible. Challenges SHOULD therefore be at least 16 bytes long.

§ 13.4.4. Attestation Limitations

This section is not normative.

When registering a new credential, the attestation statement, if present, may allow the WebAuthn Relying Party to derive assurances about various authenticator qualities. For example, the authenticator model, or how it stores and protects credential private keys. However, it is important to note that an attestation statement, on its own, provides no means for a Relying Party to verify that an attestation object was generated by the authenticator the user intended, and not by a man-in-the-middle attacker. For example, such an attacker could use malicious code injected into Relying Party script. The Relying Party must therefore rely on other means, e.g., TLS and related technologies, to protect the attestation object from man-in-the-middle attacks.

Under the assumption that a registration ceremony is completed securely, and that the authenticator maintains confidentiality of the credential private key, subsequent authentication ceremonies using that public key credential are resistant to man-in-the-middle attacks.

The discussion above holds for all attestation types. In all cases it is possible for a man-in-the-middle attacker to replace the `PublicKeyCredential` object, including the attestation statement and the credential public key to be registered, and subsequently tamper with future authentication assertions scoped for the same Relying Party and passing through the same attacker.

Such an attack would potentially be detectable; since the Relying Party has registered the attacker's credential public key rather than the user's, the attacker must tamper with all subsequent authentication ceremonies with that Relying Party: unscathed ceremonies will fail, potentially revealing the attack.

Attestation types other than Self Attestation and None can increase the difficulty of such attacks, since Relying Parties can possibly display authenticator information, e.g., model designation, to the user. An attacker might therefore need to use a genuine authenticator of the same model as the user's authenticator, or the user might notice that the Relying Party reports a different authenticator model than the user expects.

> Note: All variants of man-in-the-middle attacks described above are more difficult for an attacker to mount than a man-in-the-middle attack against conventional password authentication.

§ 13.4.5. Revoked Attestation Certificates

If attestation certificate validation fails due to a revoked intermediate attestation CA certificate, and the Relying Party's policy requires rejecting the registration/authentication request in these situations, then it is RECOMMENDED that the Relying Party also un-registers (or marks with a trust level equivalent to "self attestation") public key credentials that were registered after the CA compromise date using an attestation certificate chaining up to the same intermediate CA. It is thus

RECOMMENDED that Relying Parties remember intermediate attestation CA certificates during registration in order to un-register related public key credentials if the registration was performed after revocation of such certificates.

See also the related security consideration for authenticators in § 13.3.2 Attestation Certificate and Attestation Certificate CA Compromise.

§ 13.4.6. Credential Loss and Key Mobility

This specification defines no protocol for backing up credential private keys, or for sharing them between authenticators. In general, it is expected that a credential private key never leaves the authenticator that created it. Losing an authenticator therefore, in general, means losing all credentials bound to the lost authenticator, which could lock the user out of an account if the user has only one credential registered with the Relying Party. Instead of backing up or sharing private keys, the Web Authentication API allows registering multiple credentials for the same user. For example, a user might register platform credentials on frequently used client devices, and one or more roaming credentials for use as backup and with new or rarely used client devices.

Relying Parties SHOULD allow and encourage users to register multiple credentials to the same account. Relying Parties SHOULD make use of the `excludeCredentials` and `user.id` options to ensure that these different credentials are bound to different authenticators.

§ 13.4.7. Unprotected account detection

This section is not normative.

This security consideration applies to Relying Parties that support authentication ceremonies with a non-empty `allowCredentials` argument as the first authentication step. For example, if using authentication with server-side credentials as the first authentication step.

In this case the `allowCredentials` argument risks leaking information about which user accounts have WebAuthn credentials registered and which do not, which may be a signal of account protection strength. For example, say an attacker can initiate an authentication ceremony by providing only a username, and the Relying Party responds with an non-empty `allowCredentials` for some users, and with failure or a password challenge for other users. The attacker can then conclude that the latter user accounts likely do not require a WebAuthn assertion for successful authentication, and thus focus an attack on those likely weaker accounts.

This issue is similar to the one described in § 14.6.2 Username Enumeration and § 14.6.3 Privacy leak via credential IDs, and can be mitigated in similar ways.

§ 14. Privacy Considerations

The privacy principles in [FIDO-Privacy-Principles] also apply to this specification.

This section is divided by audience; general privacy considerations are direct subsections of this section, while privacy considerations specifically for authenticator, client and Relying Party implementers are grouped into respective subsections.

§ 14.1. De-anonymization Prevention Measures

This section is not normative.

Many aspects of the design of the Web Authentication API are motivated by privacy concerns. The main concern considered in this specification is the protection of the user's personal identity, i.e., the identification of a human being or a correlation of separate identities as belonging to the same human being. Although the Web Authentication API does not use or provide any form of global identity, the following kinds of potentially correlatable identifiers are used:

- The user's credential IDs and credential public keys.

These are registered by the WebAuthn Relying Party and subsequently used by the user to prove possession of the corresponding credential private key. They are also visible to the client in the communication with the authenticator.

- The user's identities specific to each Relying Party, e.g., usernames and user handles.

  These identities are obviously used by each Relying Party to identify a user in their system. They are also visible to the client in the communication with the authenticator.

- The user's biometric characteristic(s), e.g., fingerprints or facial recognition data [ISOBiometricVocabulary].

  This is optionally used by the authenticator to perform user verification. It is not revealed to the Relying Party, but in the case of platform authenticators, it might be visible to the client depending on the implementation.

- The models of the user's authenticators, e.g., product names.

  This is exposed in the attestation statement provided to the Relying Party during registration. It is also visible to the client in the communication with the authenticator.

- The identities of the user's authenticators, e.g., serial numbers.

  This is possibly used by the client to enable communication with the authenticator, but is not exposed to the Relying Party.

Some of the above information is necessarily shared with the Relying Party. The following sections describe the measures taken to prevent malicious Relying Parties from using it to discover a user's personal identity.

§ 14.2. Anonymous, Scoped, Non-correlatable Public Key Credentials

This section is not normative.

Although Credential IDs and credential public keys are necessarily shared with the WebAuthn Relying Party to enable strong authentication, they are designed to be minimally identifying and not shared between Relying Parties.

- Credential IDs and credential public keys are meaningless in isolation, as they only identify credential key pairs and not users directly.
- Each public key credential is strictly scoped to a specific Relying Party, and the client ensures that its existence is not revealed to other Relying Parties. A malicious Relying Party thus cannot ask the client to reveal a user's other identities.
- The client also ensures that the existence of a public key credential is not revealed to the Relying Party without user consent. This is detailed further in § 14.5.1 Registration Ceremony Privacy and § 14.5.2 Authentication Ceremony Privacy. A malicious Relying Party thus cannot silently identify a user, even if the user has a public key credential registered and available.
- Authenticators ensure that the credential IDs and credential public keys of different public key credentials are not correlatable as belonging to the same user. A pair of malicious Relying Parties thus cannot correlate users between their systems without additional information, e.g., a willfully reused username or e-mail address.
- Authenticators ensure that their attestation certificates are not unique enough to identify a single authenticator or a small group of authenticators. This is detailed further in § 14.4.1 Attestation Privacy. A pair of malicious Relying Parties thus cannot correlate users between their systems by tracking individual authenticators.

Additionally, a client-side discoverable public key credential source can optionally include a user handle specified by the Relying Party. The credential can then be used to both identify and authenticate the user. This means that a privacy-conscious Relying Party can allow the user to create an account without a traditional username, further improving non-correlatability between Relying Parties.

§ 14.3. Authenticator-local Biometric Recognition

Biometric authenticators perform the biometric recognition internally in the authenticator - though for platform authenticators the biometric data might also be visible to the client, depending on the implementation. Biometric data is not revealed to the WebAuthn Relying Party; it is used only locally to perform user verification authorizing the creation and registration of, or authentication using, a public key credential. A malicious Relying Party therefore cannot discover the

user's personal identity via biometric data, and a security breach at a Relying Party cannot expose biometric data for an attacker to use for forging logins at other Relying Parties.

In the case where a Relying Party requires biometric recognition, this is performed locally by the biometric authenticator perfoming user verification and then signaling the result by setting the UV flag in the signed assertion response, instead of revealing the biometric data itself to the Relying Party.

§ 14.4. Privacy considerations for authenticators

§ 14.4.1. Attestation Privacy

Attestation certificates and attestation key pairs can be used to track users or link various online identities of the same user together. This can be mitigated in several ways, including:

- A WebAuthn Authenticator manufacturer may choose to ship authenticators in batches where authenticators in a batch share the same attestation certificate (called Basic Attestation or batch attestation). This will anonymize the user at the risk of not being able to revoke a particular attestation certificate if its private key is compromised. The authenticator manufacturer SHOULD then ensure that such batches are large enough to provide meaningful anonymization, while also minimizing the batch size in order to limit the number of affected users in case an attestation private key is compromised.

  [UAFProtocol] requires that at least 100,000 authenticator devices share the same attestation certificate in order to produce sufficiently large groups. This may serve as guidance about suitable batch sizes.

- A WebAuthn Authenticator may be capable of dynamically generating different attestation key pairs (and requesting related certificates) per-credential as described in the Anonymization CA approach. For example, an authenticator can ship with a master attestation private key (and certificate), and combined with a cloud-operated Anonymization CA, can dynamically generate per-credential attestation key pairs and attestation certificates.

> Note: In various places outside this specification, the term "Privacy CA" is used to refer to what is termed here as an Anonymization CA. Because the Trusted Computing Group (TCG) also used the term "Privacy CA" to refer to what the TCG now refers to as an Attestation CA (ACA) [TCG-CMCProfile-AIKCertEnroll], we are using the term Anonymization CA here to try to mitigate confusion in the specific context of this specification.

§ 14.4.2. Privacy of personally identifying information Stored in Authenticators

Authenticators MAY provide additional information to clients outside what's defined by this specification, e.g., to enable the client to provide a rich UI with which the user can pick which credential to use for an authentication ceremony. If an authenticator chooses to do so, it SHOULD NOT expose personally identifying information unless successful user verification has been performed. If the authenticator supports user verification with more than one concurrently enrolled user, the authenticator SHOULD NOT expose personally identifying information of users other than the currently verified user. Consequently, an authenticator that is not capable of user verification SHOULD NOT store personally identifying information.

For the purposes of this discussion, the user handle conveyed as the `id` member of `PublicKeyCredentialUserEntity` is not considered personally identifying information; see § 14.6.1 User Handle Contents.

These recommendations serve to prevent an adversary with physical access to an authenticator from extracting personally identifying information about the authenticator's enrolled user(s).

§ 14.5. Privacy considerations for clients

§ 14.5.1. Registration Ceremony Privacy

In order to protect users from being identified without consent, implementations of the `[[Create]](origin, options, sameOriginWithAncestors)` method need to take care to not leak information that could enable a malicious WebAuthn Relying Party to distinguish between these cases, where "excluded" means that at least one of the credentials listed by the Relying Party in `excludeCredentials` is bound to the authenticator:

- No authenticators are present.
- At least one authenticator is present, and at least one present authenticator is excluded.

If the above cases are distinguishable, information is leaked by which a malicious Relying Party could identify the user by probing for which credentials are available. For example, one such information leak is if the client returns a failure response as soon as an excluded authenticator becomes available. In this case - especially if the excluded authenticator is a platform authenticator - the Relying Party could detect that the ceremony was canceled before the timeout and before the user could feasibly have canceled it manually, and thus conclude that at least one of the credentials listed in the `excludeCredentials` parameter is available to the user.

The above is not a concern, however, if the user has consented to create a new credential before a distinguishable error is returned, because in this case the user has confirmed intent to share the information that would be leaked.

§ 14.5.2. Authentication Ceremony Privacy

In order to protect users from being identified without consent, implementations of the `[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)` method need to take care to not leak information that could enable a malicious WebAuthn Relying Party to distinguish between these cases, where "named" means that the credential is listed by the Relying Party in `allowCredentials`:

- A named credential is not available.
- A named credential is available, but the user does not consent to use it.

If the above cases are distinguishable, information is leaked by which a malicious Relying Party could identify the user by probing for which credentials are available. For example, one such information leak is if the client returns a failure response as soon as the user denies consent to proceed with an authentication ceremony. In this case the Relying Party could detect that the ceremony was canceled by the user and not the timeout, and thus conclude that at least one of the credentials listed in the `allowCredentials` parameter is available to the user.

14.5.3. Privacy Between Operating System Accounts

If a platform authenticator is included in a client device with a multi-user operating system, the platform authenticator and client device SHOULD work together to ensure that the existence of any platform credential is revealed only to the operating system user that created that platform credential.

14.6. Privacy considerations for Relying Parties

14.6.1. User Handle Contents

Since the user handle is not considered personally identifying information in § 14.4.2 Privacy of personally identifying information Stored in Authenticators, the Relying Party MUST NOT include personally identifying information, e.g., e-mail addresses or usernames, in the user handle. This includes hash values of personally identifying information, unless the hash function is salted with salt values private to the Relying Party, since hashing does not prevent probing for guessable input values. It is RECOMMENDED to let the user handle be 64 random bytes, and store this value in the user's account.

§ 14.6.2. Username Enumeration

While initiating a registration or authentication ceremony, there is a risk that the WebAuthn Relying Party might leak sensitive information about its registered users. For example, if a Relying Party uses e-mail addresses as usernames and an attacker attempts to initiate an authentication ceremony for "alex.mueller@example.com" and the Relying Party responds with a failure, but then successfully initiates an authentication ceremony for "j.doe@example.com", then the attacker can conclude that "j.doe@example.com" is registered and "alex.mueller@example.com" is not. The Relying Party has thus leaked the possibly sensitive information that "j.doe@example.com" has an account at this Relying Party.

The following is a non-normative, non-exhaustive list of measures the Relying Party may implement to mitigate or prevent information leakage due to such an attack:

- For registration ceremonies:
  - If the Relying Party uses Relying Party-specific usernames to identify users:
    - When initiating a registration ceremony, disallow registration of usernames that are syntactically valid e-mail addresses.

      > Note: The motivation for this suggestion is that in this case the Relying Party probably has no choice but to fail the registration ceremony if the user attempts to register a username that is already registered, and an information leak might therefore be unavoidable. By disallowing e-mail addresses as usernames, the impact of the leakage can be mitigated since it will be less likely that a user has the same username at this Relying Party as at other Relying Parties.
  - If the Relying Party uses e-mail addresses to identify users:
    - When initiating a registration ceremony, interrupt the user interaction after the e-mail address is supplied and send a message to this address, containing an unpredictable one-time code and instructions for how to use it

to proceed with the ceremony. Display the same message to the user in the web interface regardless of the contents of the sent e-mail and whether or not this e-mail address was already registered.

> Note: This suggestion can be similarly adapted for other externally meaningful identifiers, for example, national ID numbers or credit card numbers — if they provide similar out-of-band contact information, for example, conventional postal address.

- For authentication ceremonies:
  - If, when initiating an authentication ceremony, there is no account matching the provided username, continue the ceremony by invoking `navigator.credentials.get()` using a syntactically valid `PublicKeyCredentialRequestOptions` object that is populated with plausible imaginary values.

    This approach could also be used to mitigate information leakage via `allowCredentials`; see § 13.4.7 Unprotected account detection and § 14.6.3 Privacy leak via credential IDs.

    > Note: The username may be "provided" in various Relying Party-specific fashions: login form, session cookie, etc.

    > Note: If returned imaginary values noticeably differ from actual ones, clever attackers may be able to discern them and thus be able to test for existence of actual accounts. Examples of noticeably different values include if the values are always the same for all username inputs, or are different in repeated attempts with the same username input. The `allowCredentials` member could therefore be populated with pseudo-random values derived deterministically from the username, for example.

  - When verifying an `AuthenticatorAssertionResponse` response from the authenticator, make it indistinguishable whether verification failed because the signature is invalid or because no such user or credential is registered.

- Perform a multi-step authentication ceremony, e.g., beginning with supplying username and password or a session cookie, before initiating the WebAuthn ceremony as a subsequent step. This moves the username enumeration problem from the WebAuthn step to the preceding authentication step, where it may be easier to solve.

§ 14.6.3. Privacy leak via credential IDs

This section is not normative.

This privacy consideration applies to Relying Parties that support authentication ceremonies with a non-empty `allowCredentials` argument as the first authentication step. For example, if using authentication with server-side credentials as the first authentication step.

In this case the `allowCredentials` argument risks leaking personally identifying information, since it exposes the user's credential IDs to an unauthenticated caller. Credential IDs are designed to not be correlatable between Relying Parties, but the length of a credential ID might be a hint as to what type of authenticator created it. It is likely that a user will use the same username and set of authenticators for several Relying Parties, so the number of credential IDs in `allowCredentials` and their lengths might serve as a global correlation handle to de-anonymize the user. Knowing a user's credential IDs also makes it possible to confirm guesses about the user's identity given only momentary physical access to one of the user's authenticators.

In order to prevent such information leakage, the Relying Party could for example:

- Perform a separate authentication step, such as username and password authentication or session cookie authentication, before initiating the WebAuthn authentication ceremony and exposing the user's credential IDs.
- Use client-side discoverable credentials, so the `allowCredentials` argument is not needed.

If the above prevention measures are not available, i.e., if `allowCredentials` needs to be exposed given only a username, the Relying Party could mitigate the privacy leak using the same approach of returning imaginary credential IDs as discussed in § 14.6.2 Username Enumeration.

§ 15. Accessibility Considerations

User verification-capable authenticators, whether roaming or platform, should offer users more than one user verification method. For example, both fingerprint sensing and PIN entry. This allows for fallback to other user verification means if the selected one is not working for some reason. Note that in the case of roaming authenticators, the authenticator and platform might work together to provide a user verification method such as PIN entry [FIDO-CTAP].

Relying Parties, at registration time, SHOULD provide affordances for users to complete future authorization gestures correctly. This could involve naming the authenticator, choosing a picture to associate with the device, or entering freeform text instructions (e.g., as a reminder-to-self).

Ceremonies relying on timing, e.g., a registration ceremony (see `timeout`) or an authentication ceremony (see `timeout`), ought to follow [WCAG21]'s Guideline 2.2 Enough Time. If a client platform determines that a Relying Party-supplied timeout does not appropriately adhere to the latter [WCAG21] guidelines, then the client platform MAY adjust the timeout accordingly.

§ 16. Acknowledgements

We thank the following people for their reviews of, and contributions to, this specification: Yuriy Ackermann, James Barclay, Richard Barnes, Dominic Battré, Julien Cayzac, Domenic Denicola, Rahul Ghosh, Brad Hill, Jing Jin, Wally Jones, Ian Kilpatrick, Axel Nennker, Yoshikazu Nojima, Kimberly Paulhamus, Adam Powers, Yaron Sheffer, Ki-Eun Shin, Anne van Kesteren, Johan Verrept, and Boris Zbarsky.

Thanks to Adam Powers for creating the overall registration and authentication flow diagrams (Figure 1 and Figure 2).

We thank Anthony Nadalin, John Fontana, and Richard Barnes for their contributions as co-chairs of the Web Authentication Working Group.

We also thank Wendy Seltzer, Samuel Weiler, and Harry Halpin for their contributions as our W3C Team Contacts.

§ Index

§ Terms defined by this specification

§ Terms defined by reference

[BCP47] defines the following terms:

language tag

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

- Credential
- CredentialCreationOptions
- CredentialRequestOptions
- CredentialsContainer
- Request a Credential
- [[CollectFromCredentialStore]](origin, options, sameOriginWithAncestors)
- [[Create]](origin, options, sameOriginWithAncestors)
- [[Store]](credential, sameOriginWithAncestors)
- [[discovery]]
- [[type]]
- create()
- credential
- credential source
- get()
- id
- remote
- same-origin with its ancestors
- signal (for CredentialRequestOptions)
- store()
- type
- user mediation

[DOM4] defines the following terms:

- AbortController
- Document
- aborted flag
- document

[ECMAScript] defines the following terms:

- %arraybuffer%
- internal method
- internal slot
- own property

[ENCODING] defines the following terms:

- utf-8 decode
- utf-8 encode

[FETCH] defines the following terms:

- window

[FIDO-APPID] defines the following terms:

- determining if a caller's facetid is authorized for an appid
- determining the facetid of a calling application

[FIDO-CTAP] defines the following terms:

- ctap2 canonical cbor encoding form
- large, per-credential blobs
- §6.2. responses

[FIDO-Registry] defines the following terms:

- section 3.1 user verification methods
- section 3.2 key protection types
- section 3.3 matcher protection types
- section 3.6.2 public key representation formats

[FIDO-U2F-Message-Formats] defines the following terms:

- application parameter
- section 4.3
- section 5.4

[FileAPI] defines the following terms:

- object

[HTML] defines the following terms:

- allow
- allowed to use
- ascii serialization of an origin
- browsing context
- current settings object
- document.domain
- effective domain
- environment settings object
- global object
- iframe
- in parallel
- is a registrable domain suffix of or is equal to
- is not a registrable domain suffix of and is not equal to
- opaque origin
- origin (for environment settings object)
- permissions policy
- relevant settings object
- tuple origin

[INFRA] defines the following terms:

- append (for set)
- boolean
- byte sequence
- continue
- empty
- exist
- for each (for map)
- is empty
- is not empty
- item (for struct)
- list
- map
- ordered set
- remove
- serialize json to bytes
- set (for map)
- size
- struct
- while
- willful violation

[page-visibility] defines the following terms:

- visibility states

[Permissions-Policy] defines the following terms:

- default allowlist
- policy-controlled feature

[RFC4949] defines the following terms:

- man-in-the-middle attack
- salt
- salted

[RFC5280] defines the following terms:

- subjectpublickeyinfo

[RFC8152] defines the following terms:

- cose key
- crv
- kty
- section 13.1
- section 7
- section 8.1

[RFC8230] defines the following terms:

- section 2
- section 4

[RFC8610] defines the following terms:

- group sockets

[secure-contexts] defines the following terms:

- secure contexts

[SP800-800-63r3] defines the following terms:

- authentication factor
- multi-factor
- second-factor
- single-factor
- something you are
- something you have
- something you know

[TokenBinding] defines the following terms:

- token binding
- token binding id

[URL] defines the following terms:

- domain
- empty host
- host
- ipv4 address
- ipv6 address
- opaque host
- port
- scheme
- valid domain
- valid domain string

[UTR29] defines the following terms:

- grapheme cluster

[WebDriver] defines the following terms:

- endpoint node
- extension capability
- extension command
- getting a property
- invalid argument
- matching capabilities
- remote end steps
- set a property
- success
- unsupported operation
- validating capabilities
- webdriver error
- webdriver error code

[WebIDL] defines the following terms:

- AbortError
- ArrayBuffer
- BufferSource
- ConstraintError
- DOMException
- DOMString
- Exposed
- InvalidStateError
- NotAllowedError
- NotSupportedError
- Promise
- SameObject
- SecureContext
- SecurityError
- TypeError
- USVString
- UnknownError
- boolean
- get a copy of the bytes held by the buffer source
- interface object
- long
- sequence
- unsigned long

[whatwg html] defines the following terms:

- focus

[whatwg url] defines the following terms:

- same site

§ References

§ Normative References

[BCP47]

A. Phillips; M. Davis. Tags for Identifying Languages. September 2009. IETF Best Current Practice. URL: https://tools.ietf.org/html/bcp47

[CREDENTIAL-MANAGEMENT-1]

Mike West. Credential Management Level 1. 17 January 2019. WD. URL: https://www.w3.org/TR/credential-management-1/

[DOM4]

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

[ECMAScript]

ECMAScript Language Specification. URL: https://tc39.es/ecma262/

[ENCODING]

Anne van Kesteren. Encoding Standard. Living Standard. URL: https://encoding.spec.whatwg.org/

[FETCH]

Anne van Kesteren. Fetch Standard. Living Standard. URL: https://fetch.spec.whatwg.org/

[FIDO-APPID]

D. Balfanz; et al. FIDO AppID and Facet Specification. 27 February 2018. FIDO Alliance Implementation Draft. URL: https://fidoalliance.org/specs/fido-v2.0-id-20180227/fido-appid-and-facets-v2.0-id-20180227.html

[FIDO-CTAP]

M. Antoine; et al. Client to Authenticator Protocol. 27 February 2018. FIDO Alliance Implementation Draft. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html

[FIDO-Privacy-Principles]

FIDO Alliance. FIDO Privacy Principles. FIDO Alliance Whitepaper. URL: https://fidoalliance.org/wp-content/uploads/2014/12/FIDO_Alliance_Whitepaper_Privacy_Principles.pdf

[FIDO-Registry]

R. Lindemann; D. Baghdasaryan; B. Hill. FIDO Registry of Predefined Values. 27 February 2018. FIDO Alliance Implementation Draft. URL: https://fidoalliance.org/specs/fido-v2.0-id-20180227/fido-registry-v2.0-id-20180227.html

[FIDO-U2F-Message-Formats]

D. Balfanz; J. Ehrensvard; J. Lang. FIDO U2F Raw Message Formats. FIDO Alliance Implementation Draft. URL: https://fidoalliance.org/specs/fido-u2f-v1.1-id-20160915/fido-u2f-raw-message-formats-v1.1-id-20160915.html

[FileAPI]

Marijn Kruisselbrink; Arun Ranganathan. File API. 11 September 2019. WD. URL: https://www.w3.org/TR/FileAPI/

[HTML]

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

[IANA-COSE-ALGS-REG]

IANA CBOR Object Signing and Encryption (COSE) Algorithms Registry. URL: https://www.iana.org/assignments/cose/cose.xhtml#algorithms

[IANA-WebAuthn-Registries]

IANA. Web Authentication (WebAuthn) registries. URL: https://www.iana.org/assignments/webauthn/

[INFRA]

Anne van Kesteren; Domenic Denicola. Infra Standard. Living Standard. URL: https://infra.spec.whatwg.org/

[PAGE-VISIBILITY]

Jatinder Mann; Arvind Jain. Page Visibility (Second Edition). 29 October 2013. REC. URL: https://www.w3.org/TR/page-visibility/

[Permissions-Policy]

Ian Clelland. Permissions Policy. 16 July 2020. WD. URL: https://www.w3.org/TR/permissions-policy-1/

[RFC2119]

S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC3986]

T. Berners-Lee; R. Fielding; L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. January 2005. Internet Standard. URL: https://tools.ietf.org/html/rfc3986

[RFC4648]

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

[RFC4949]

R. Shirey. Internet Security Glossary, Version 2. August 2007. Informational. URL: https://tools.ietf.org/html/rfc4949

[RFC5234]

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

[RFC5280]

D. Cooper; et al. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. May 2008. Proposed Standard. URL: https://tools.ietf.org/html/rfc5280

[RFC5890]

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

[RFC6454]

A. Barth. The Web Origin Concept. December 2011. Proposed Standard. URL: https://tools.ietf.org/html/rfc6454

[RFC7515]

M. Jones; J. Bradley; N. Sakimura. JSON Web Signature (JWS). May 2015. Proposed Standard. URL: https://tools.ietf.org/html/rfc7515

[RFC8152]

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

[RFC8230]

M. Jones. Using RSA Algorithms with CBOR Object Signing and Encryption (COSE) Messages. September 2017. Proposed Standard. URL: https://tools.ietf.org/html/rfc8230

[RFC8264]

P. Saint-Andre; M. Blanchet. PRECIS Framework: Preparation, Enforcement, and Comparison of Internationalized Strings in Application Protocols. October 2017. Proposed Standard. URL: https://tools.ietf.org/html/rfc8264

[RFC8265]

P. Saint-Andre; A. Melnikov. Preparation, Enforcement, and Comparison of Internationalized Strings Representing Usernames and Passwords. October 2017. Proposed Standard. URL: https://tools.ietf.org/html/rfc8265

[RFC8266]

P. Saint-Andre. Preparation, Enforcement, and Comparison of Internationalized Strings Representing Nicknames. October 2017. Proposed Standard. URL: https://tools.ietf.org/html/rfc8266

[RFC8610]

H. Birkholz; C. Vigano; C. Bormann. Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures. June 2019. IETF Proposed Standard. URL: https://tools.ietf.org/html/rfc8610

[RFC8809]

Jeff Hodges; Giridhar Mandyam; Michael B. Jones. Registries for Web Authentication (WebAuthn). August 2020. IETF Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc8809

[RFC8949]

C. Bormann; P. Hoffman. Concise Binary Object Representation (CBOR). December 2020. Internet Standard. URL: https://tools.ietf.org/html/rfc8949

[SEC1]

SEC1: Elliptic Curve Cryptography, Version 2.0. URL: http://www.secg.org/sec1-v2.pdf

[SECURE-CONTEXTS]

Mike West. Secure Contexts. 15 September 2016. CR. URL: https://www.w3.org/TR/secure-contexts/

[SP800-800-63r3]

Paul A. Grassi; Michael E. Garcia; James L. Fenton. NIST Special Publication 800-63: Digital Identity Guidelines. June 2017. URL: https://pages.nist.gov/800-63-3/sp800-63-3.html

[TCG-CMCProfile-AIKCertEnroll]

Scott Kelly; et al. TCG Infrastructure Working Group: A CMC Profile for AIK Certificate Enrollment. 24 March 2011. Published. URL: https://trustedcomputinggroup.org/wp-content/uploads/IWG_CMC_Profile_Cert_Enrollment_v1_r7.pdf

[TokenBinding]

A. Popov; et al. The Token Binding Protocol Version 1.0. October, 2018. IETF Proposed Standard. URL: https://tools.ietf.org/html/rfc8471

[TPMv2-EK-Profile]

TCG EK Credential Profile for TPM Family 2.0. URL: https://www.trustedcomputinggroup.org/wp-content/uploads/Credential_Profile_EK_V2.0_R14_published.pdf

[TPMv2-Part1]

Trusted Platform Module Library, Part 1: Architecture. URL: https://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-1-Architecture-01.38.pdf

[TPMv2-Part2]

Trusted Platform Module Library, Part 2: Structures. URL: https://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-2-Structures-01.38.pdf

[TPMv2-Part3]

Trusted Platform Module Library, Part 3: Commands. URL: https://www.trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-3-Commands-01.38.pdf

[URL]

Anne van Kesteren. URL Standard. Living Standard. URL: https://url.spec.whatwg.org/

[UTR29]

UNICODE Text Segmentation. URL: http://www.unicode.org/reports/tr29/

[WCAG21]

Andrew Kirkpatrick; et al. Web Content Accessibility Guidelines (WCAG) 2.1. 5 June 2018. REC. URL: https://www.w3.org/TR/WCAG21/

[WebDriver]

Simon Stewart; David Burns. WebDriver. 5 June 2018. REC. URL: https://www.w3.org/TR/webdriver1/

[WebIDL]

Boris Zbarsky. Web IDL. 15 December 2016. ED. URL: https://heycam.github.io/webidl/

§ Informative References

[Ceremony]

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

[CSS-OVERFLOW-3]

David Baron; Elika Etemad; Florian Rivoal. CSS Overflow Module Level 3. 3 June 2020. WD. URL: https://www.w3.org/TR/css-overflow-3/

[EduPersonObjectClassSpec]

EduPerson. ongoing. URL: https://refeds.org/eduperson

[FIDO-Transports-Ext]

FIDO Alliance. FIDO U2F Authenticator Transports Extension. FIDO Alliance Proposed Standard. URL: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-authenticator-transports-extension-v1.2-ps-20170411.html

[FIDO-UAF-AUTHNR-CMDS]

R. Lindemann; J. Kemp. FIDO UAF Authenticator Commands. FIDO Alliance Implementation Draft. URL: https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-authnr-cmds-v1.1-id-20170202.html

[FIDOAuthnrSecReqs]

D. Biggs; et al. FIDO Authenticator Security Requirements. FIDO Alliance Final Documents. URL: https://fidoalliance.org/specs/fido-security-requirements-v1.0-fd-20170524/

[FIDOMetadataService]

R. Lindemann; B. Hill; D. Baghdasaryan. FIDO Metadata Service. 27 February 2018. FIDO Alliance Implementation Draft. URL: https://fidoalliance.org/specs/fido-v2.0-id-20180227/fido-metadata-service-v2.0-id-20180227.html

[FIDOSecRef]

R. Lindemann; et al. FIDO Security Reference. 27 February 2018. FIDO Alliance Implementation Draft. URL: https://fidoalliance.org/specs/fido-v2.0-id-20180227/fido-security-ref-v2.0-id-20180227.html

[FIDOU2FJavaScriptAPI]

D. Balfanz; A. Birgisson; J. Lang. FIDO U2F JavaScript API. FIDO Alliance Proposed Standard. URL: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-javascript-api-v1.2-ps-20170411.html

[ISOBiometricVocabulary]

ISO/IEC JTC1/SC37. Information technology — Vocabulary — Biometrics. 15 December 2012. International Standard: ISO/IEC 2382-37:2012(E) First Edition. URL: http://standards.iso.org/ittf/PubliclyAvailableStandards/c055194_ISOIEC_2382-37_2012.zip

[RFC3279]

L. Bassham; W. Polk; R. Housley. Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. April 2002. Proposed Standard. URL: https://tools.ietf.org/html/rfc3279

[RFC5958]

S. Turner. Asymmetric Key Packages. August 2010. Proposed Standard. URL: https://tools.ietf.org/html/rfc5958

[RFC6265]

A. Barth. HTTP State Management Mechanism. April 2011. Proposed Standard. URL: https://httpwg.org/specs/rfc6265.html

[RFC8017]

K. Moriarty, Ed.; et al. PKCS #1: RSA Cryptography Specifications Version 2.2. November 2016. Informational. URL: https://tools.ietf.org/html/rfc8017

[UAFProtocol]

R. Lindemann; et al. FIDO UAF Protocol Specification v1.0. FIDO Alliance Proposed Standard. URL: https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-protocol-v1.0-ps-20141208.html

[WebAuthn-1]

Dirk Balfanz; et al. Web Authentication:An API for accessing Public Key Credentials Level 1. 4 March 2019. REC. URL: https://www.w3.org/TR/webauthn-1/

[WebAuthnAPIGuide]

Web Authentication API Guide. Experimental. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Authentication_API

§ IDL Index

§ Issues Index

ISSUE 1 The WHATWG HTML WG is discussing whether to provide a hook when a browsing context gains or loses focuses. If a hook is provided, the above paragraph will be updated to include the hook. See WHATWG HTML WG Issue #2711 for more details. ↵