Client to Authenticator Protocol (CTAP)

Review Draft, October 03, 2024

REVIEW DRAFT

This version:
https://fidoalliance.org/specs/fido-v2.2-rd-20241003/fido-client-to-authenticator-protocol-v2.2-rd-20241003.html

Previous Versions:
https://fidoalliance.org/specs/fido-v2.1-ps-20210615/

Issue Tracking:
GitHub

Editors:
John Bradley (Yubico)
Michael B. Jones (independent)
Akshay Kumar (Microsoft)
Rolf Lindemann (Nok Nok Labs)
Johan Verrept (OneSpan)
David Waite (Ping Identity)

Former Editors:
Matthieu Antoine (Gemalto)
Vijay Bharadwaj (Microsoft)
Arnar Birgisson (Google)
Christiaan Brand (Google)
Alexei Czeskis (Google)
Thomas Duboucher (Thales Group)
Jakob Ehrensvärd (Yubico)
Jeff Hodges (Google)
Mirko J. Ploch (SurePassID)
Adam Powers (FIDO Alliance)

Contributors:
Chad Armstrong (Google)
Tim Cappalli (Okta)
Konstantinos Georgantas (Yubico)
Fabian Kaczmarczyck (Google)
Kim Paulhamus (Google)
Nina Satragno (Google)
Nuno Sung (AuthenTrend)

Abstract

This specification describes an application layer protocol for communication between a roaming authenticator and another client/platform, as well as bindings of this application protocol to a variety of transport protocols using different physical media. The application layer protocol defines requirements for such transport protocols. Each transport binding defines the details of how such transport layer connections should be set up, in a manner that meets the requirements of the application layer protocol.

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 FIDO Alliance publications and the latest revision of this technical report can be found in the FIDO Alliance specifications index at https://fidoalliance.org/specifications/.

This document was published by the FIDO Alliance as a Review Draft Specification. This document is intended to become a FIDO Alliance Proposed Standard. If you wish to make comments regarding this document, please Contact Us. All comments are welcome.

This is a Review Draft Specification and is not intended to be a basis for any implementations as the Specification may change. Permission is hereby granted to use the Specification solely for the purpose of reviewing the Specification. No rights are granted to prepare derivative works of this Specification. Entities seeking permission to reproduce portions of this Specification for other uses must contact the FIDO Alliance to determine whether an appropriate license for such use is available.

Implementation of certain elements of this Specification may require licenses under third party intellectual property rights, including without limitation, patent rights. The FIDO Alliance, Inc. and its Members and any other contributors to the Specification are not, and shall not be held, responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

THIS FIDO ALLIANCE SPECIFICATION IS PROVIDED “AS IS” AND WITHOUT ANY WARRANTY OF ANY

KIND, INCLUDING, WITHOUT LIMITATION, ANY EXPRESS OR IMPLIED WARRANTY OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Table of Contents

1. Introduction§

This section is not normative.

This protocol is intended to be used in scenarios where a user interacts with a **Relying Party** (a website or native app) on some platform (e.g., a PC) which prompts the user to interact with a roaming authenticator (e.g., a smartphone).

In order to provide evidence of user interaction, a roaming authenticator implementing this protocol may have a built-in mechanism to obtain a "user gesture", allowing the platform to collect a PIN on behalf of the authenticator.

1.1. Relationship to Other Specifications

This specification is part of the FIDO2 project, which includes this specification and is related to the W3C [WebAuthn] specification. This specification refers to two CTAP protocol versions:

1. The CTAP1/U2F protocol, which is defined by the U2F Raw Messages specification [U2FRawMsgs]. CTAP1/U2F messages are recognizable by their APDU-like binary structure. CTAP1/U2F may also be referred to as CTAP 1.2 or U2F 1.2. The latter was the U2F specification version used as the basis for several portions of this specification. Authenticators implementing CTAP1/U2F are typically referred to as U2F authenticators or CTAP1 authenticators.
2. The CTAP2 protocol, whose messages are encoded in the CTAP2 canonical CBOR encoding form. Authenticators implementing CTAP2 are referred to as CTAP2 authenticators, FIDO2 authenticators, or WebAuthn authenticators.

Both CTAP1 and CTAP2 share the same underlying transports: USB Human Interface Device (USB HID), Near Field Communication (NFC), and Bluetooth Smart / Bluetooth Low Energy Technology (BLE).

Whole documents or specific features may be **superseded** by this document. A superseded document or feature MAY be implemented if optional, but it exists purely for backwards compatibility with older platforms or authenticators. Thus a superseded document or feature SHOULD NOT be used unless the replacement is not implemented by the counterparty. (Superseded features are not automatically optional, e.g. a CTAP 2.1 authenticator MUST still support authenticatorClientPIN's getPinToken subcommand if it supports clientPIN and CTAP 2.0.)

The [U2FUsbHid], [U2FNfc], [U2FBle], and [U2FRawMsgs] specifications, specifically, are superseded by this specification.

CTAP2 authenticators SHOULD also implement CTAP1/U2F. See § 10 Interoperating with CTAP1/U2F authenticators for details on how these protocols interoperate from the perspective of authenticators, platforms, and RPs.

Occasionally, the term "CTAP" may be used without clarifying whether it is referring to CTAP1 or CTAP2. In such cases, it should be understood to be referring to the entirety of this specification or portions of this specification that are not specific to either CTAP1 or CTAP2. For example, some error messages begin with the term "CTAP" without clarifying whether they are CTAP1- or CTAP2-specific because they are applicable to both CTAP protocol versions. CTAP protocol-specific error messages are prefixed with either "CTAP1" or "CTAP2" as appropriate.

> Note: For certifications, other requirements than those specified in this specification may apply, for example with respect to security and privacy requirements. Those seeking authenticator certifications can refer to the applicable certification documentation, from the certifying organization in question (e.g., the FIDO Alliance, FIPS, Common Criteria, etc.), for additional information and requirements.
> In particular, see here for FIDO Alliance's certification programs.

1.2. Data Elements Referenced by Other Specifications

The following data elements might be referenced by other specifications and hence should not be changed in their fundamental data type or high-level semantics without liaising with the other specifications:

1. aaguid, data type byte string and identifying the authenticator model, i.e. identical values mean that they refer to the same authenticator model and different values mean they refer to different authenticator models.
2. RP ID, data type string representing the Relying party identifier, i.e. identical values mean that they refer to the same Relying Party.
3. **credentialID**, data type byte string identifying a specific public key credential source, i.e. identical values mean that they refer to the same credential and different values mean they refer to different credentials. Note that there might be a very small probability that different credentials get assigned the same credentialID.
4. up and uv, data type boolean indicating whether user presence (up) or user verification (uv) was performed by the authenticator.

> NOTE: Some of the data elements might have an internal structure that might change. Other specifications shall not rely on such internal structure.

2. Conformance§

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

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

Authenticators and Platforms may implement additional constraints on these specifications to meet the certification requirements of programs like [CMVP], [CSPN], and [CommonCriteria].

3. Protocol Structure§

This protocol is specified in three parts:

- **Authenticator API**: At this level of abstraction, each **authenticator operation** is defined similarly to an API call - it accepts input parameters and returns either an output or error code. Note that this API level is conceptual and does not represent actual APIs. The actual APIs will be provided by each implementing platform.
- **Message Encoding**: In order to invoke a method in the authenticator API, the host must construct and encode a request and send it to the authenticator over the chosen transport protocol. The authenticator will then process the request and return an encoded response.
- **Transport-specific Binding**: Requests and responses are conveyed to roaming authenticators over specific transports (e.g., USB, NFC, Bluetooth). For each transport technology, message bindings are specified for this protocol.

This document specifies all three of the above pieces for roaming FIDO2 authenticators.

4. Protocol Overview§

The general protocol between a Relying Party application, a client platform, and an authenticator is as follows:

1. In Relying Party-oriented use cases involving credential registration or user authentication, a Relying Party application calls `navigator.credentials.create()` or `navigator.credentials.get()` if it is a website, or the client platform's equivalent API methods if it is a native application. Other use cases, such as credential management, PIN establishment/maintenance, or biometric enrollment, are typically initiated by the client platform itself.
2. The platform establishes a connection with a nominally appropriate available authenticator, having used criteria passed in by the Relying Party application and possibly other information it has to select the authenticator.
3. The platform gets information about the authenticator using the authenticatorGetInfo command, which helps it determine the authenticator's capabilities.
4. Depending upon the operation the Relying Party application, or the platform itself, initiated (in step 1), the options it supplied, and the authenticator's capabilities, the platform will invoke *one or more* further Authenticator API commands.

5. Terminology§

Built-in User Verification method
: The authenticator supports a built-in on-device user verification method like fingerprint or has a input UI with secure communication to the authenticator.

> NOTE: clientPin is not a built-in user verification method.

Credential Provider Hosting Device (CPHD)

: In the context of hybrid transports, the device running the credential provider software which interfaces with the client platform. For FIDO2 credentials, the credential provider is a passkey provider. For digital credentials (e.g. verifiable credentials), the credential provider is often a digital identity wallet.

Evidence of user interaction
: Collection of *evidence of user interaction* establishes a state of **user presence**. Also, if it is collected along with displaying a particular prompt to a user it may be considered collecting **user consent**. The general notion is that the user interacts with the authenticator in some fashion, also known as supplying a "user gesture"—e.g., touches a consent button, enters a password or a PIN, or supplies a biometric—in order to at least confirm their presence and possibly consent to some proposed action. Some "user gesture" approaches provide user verification in addition to establishing user presence, e.g., a fingerprint-based built-in user verification method.

: Platform-mediated user interactions such as clientPin may provide user verification but are not considered to assert user presence. Thus, there are transport-based affordances affecting when and for how long user

presence is established on a per-transport basis:

NFC

For authenticators without a method to collect a user gesture inside the authenticator boundary other than through a power on gesture, the act of a user placing an NFC authenticator into the NFC reader's field is considered a user gesture that establishes user presence and *provides evidence of user interaction*. This powers-up the authenticator, who then starts an ***NFC powered-up timer***, and sets an **NFC userPresent flag** to `true`. There is an associated **NFC user presence maximum time limit** of two minutes (120 seconds).

Upon the platform subsequently invoking either authenticatorMakeCredential or authenticatorGetAssertion (e.g., with the "up" option key set to 'true'):

1. If evidence of user interaction is requested then:
    1. If the platform sends a zero length pinUvAuthParam then return either CTAP2_ERR_PIN_NOT_SET if PIN is not set or CTAP2_ERR_PIN_INVALID if PIN has been set.

        > NOTE: This is done for backwards compatibility with CTAP2.0 platforms in the case where multiple authenticators are attached to the platform. In this case the authenticator must not consume the NFC userPresent flag or it will prevent authentication with some CTAP2.0 platforms.

    2. If the NFC userPresent flag's value is `true`, then consider the user as having granted permission, and set the NFC userPresent flag to `false`.
    3. Otherwise, do not consider the user as having granted permission. End the operation by returning CTAP2_ERR_UP_REQUIRED.

Upon expiry of the NFC user presence maximum time limit, the NFC userPresent flag is set to `false` if it is not already `false`.

> NOTE: This notion of user presence establishment is distinct due to the physical proximity and user action characteristics of devices employing NFC to communicate, i.e., the user placing the authenticator in the NFC field, also known as the "NFC tap". Thus, user presence is asserted even if the platform and authenticator then use a form of user verification that does not itself provide user presence, such as clientPin-based user verification (clientPin does not assert user presence when used over other transports).

For example, in an authentication scenario, the user places an NFC authenticator on an NFC reading device having a keyboard and display, and is prompted to enter a PIN. If PIN entry is completed (e.g., by pressing Enter) before the NFC user presence maximum time limit expires, the authenticator will return an assertion with the "UP" bit in authenticator data set to `true` and the NFC userPresent flag is then set to `false`.

If a user lays an NFC authenticator on an NFC reader and for whatever reason ignores it for greater than the NFC user presence maximum time limit they will need to remove the authenticator from the NFC field and re-insert it and start over to complete any interaction requiring user presence.

All other transports

If evidence of user interaction is explicitly requested (i.e., even if a pinUvAuthToken is in use) it is interactively collected at that time in an authenticator-specific manner.

pre-flight

In order to determine whether authenticatorMakeCredential's excludeList or authenticatorGetAssertion's allowList contain credential IDs that are already present on an authenticator, a platform typically invokes authenticatorGetAssertion with the "up" option key set to `false` and optionally pinUvAuthParam one or more times. If a credential is found an assertion is returned. If a valid pinUvAuthParam was also provided, the response will contain "up"=0 and "uv"=1 within the "flags bits" of the authenticator data structure, otherwise the "flag bits" will contain "up"=0 and "uv"=0.

Protected by some form of User Verification

Either or both clientPin or built-in user verification methods are supported and enabled. I.e., in the authenticatorGetInfo response the pinUvAuthToken option ID is present and set to `true`, and either clientPin option ID is present and set to `true` or uv option ID is present and set to `true` or both.

Some form of User Verification

This term refers to either clientPin or built-in user verification methods.

User action timeout

This refers to a timeout that occurs when the authenticator is waiting for direct action from the user, like a touch. (I.e. *not* a command from the platform.) The duration of this timeout is chosen by the authenticator but MUST be at least 10 seconds. Thirty seconds is a reasonable value.

6. Authenticator API§

Each operation in the authenticator API can be performed independently of the others, and all operations are asynchronous. The authenticator may enforce a limit on outstanding operations to limit resource usage - in this case, the authenticator is expected to return a busy status and the host is expected to retry the operation later.

Additionally, this protocol does not enforce in-order or reliable delivery of requests and responses; if these properties are desired, they must be provided by the underlying transport protocol or implemented at a higher layer by applications.

Note that this API level is conceptual and does not represent actual APIs. The actual APIs will be provided by each implementing platform.

Some commands or subcommands require the authenticator to maintain state. For example, the authenticatorCredentialManagement subcommand `enumerateRPsGetNextRP` implicitly assumes that the authenticator remembers which RP is next to return. The following (sub)commands require such state and are called **stateful commands**. Each such command uses and updates state that is initialized by a corresponding **state initializing command**:

1. authenticatorGetNextAssertion, with state initialized by authenticatorGetAssertion.
2. authenticatorCredentialManagement/`enumerateRPsGetNextRP`, with state initialized by enumerateRPsBegin.
3. authenticatorCredentialManagement/`enumerateCredentialsGetNextCredential`, with state initialized by enumerateCredentialsBegin.
4. authenticatorLargeBlobs where the parameter `set` is given and the parameter `offset` is non-zero, with state initialized by a prior authenticatorLargeBlobs command with `set` given and a zero `offset`.

In order to accommodate authenticators with limited capacity, the following accommodations are made:

1. The state SHOULD NOT be maintained across power cycles.
2. The authenticator MAY maintain state based on the assumption that each stateful command is exclusively preceded by either another instance of the same command, or by the corresponding state initializing command, and no more than 30 seconds will elapse between such commands. If this pattern is violated then the authenticator MAY fail any stateful command with the error CTAP2_ERR_NOT_ALLOWED. Here, “exclusively preceded” means that no other authenticator operation occurs in between. An authenticator MAY assume this globally, even when the transport-specific binding provides for independent streams of platform commands (e.g. § 11.2.3 Concurrency and channels).
3. An authenticator MUST discard the state for a stateful command command if the pinUvAuthToken that authenticated the state initializing command expires since the stateful commands do not themselves always verify a pinUvAuthToken.

The authenticator API has the following methods and data structures.

6.1. authenticatorMakeCredential (0x01)§

This method is invoked by the host to request generation of a new credential in the authenticator. It takes the following **input parameters**, several of which correspond to those defined in the authenticatorMakeCredential operation section of the Web Authentication specification:

Parameter name	Data type	Required?	Definition
clientDataHash (0x01)	Byte String	Required	Hash of the ClientData contextual binding specified by host. See [WebAuthn].
rp (0x02)	`PublicKeyCredentialRpEntity`	Required	This PublicKeyCredentialRpEntity data structure describes a Relying Party with which the new public key credential will be associated. It contains the relying party identifier (**rp.id** of type text string, (optionally) a human-friendly RP name of type text string. The RP name is to be used by the authenticator when displaying the credential to the user for selection and usage authorization. The RP name and URL are OPTIONAL so that the RP can be more privacy friendly if it chooses to. For example, for authenticators with a display, RP may not want to display name for single-factor scenarios.

Parameter name	Data type	Required?	
			NOTE: [WebAuthn-2] has removed the optional `icon` member. Authenticators MUST NOT error if the `icon` member is present, they MAY not store this value.
user (0x03)	`PublicKeyCredentialUserEntity`	Required	This PublicKeyCredentialUserEntity data structure describes the user account to which the new public key credential will be associated at the RP. It contains an RP-specific user account identifier of type byte string, (optionally) a user name of type text string, (optionally) a user display name of type text string, and (optionally) a URL of type text string, referencing a user icon image (of a user avatar, for example). Note that while an empty account identifier is valid, it has known interoperability hurdles in practice and platforms are RECOMMENDED to avoid sending them. The authenticator associates the created public key credential with the account identifier, and MAY also associate any or all of the user name, and user display name. The user name and display name are OPTIONAL for privacy reasons for single-factor scenarios where only user presence is required. For example, in certain closed physical environments like factory floors, user presence only authenticators can satisfy RP's productivity and security needs. In these environments, omitting user name and display name makes the credential more privacy friendly. Although this information is not available without user verification, devices which support user verification but do not have it configured, can be tricked into releasing this information by configuring the user verification. NOTE: [WebAuthn-2] has removed the optional `icon` member. Authenticators MUST NOT error if the `icon` member is present, they MAY not store this value.
pubKeyCredParams (0x04)	Array of `PublicKeyCredentialParameters`	Required	List of supported algorithms for credential generation, as specified in [WebAuthn]. The array is ordered from most preferred to least preferred and MUST NOT include duplicate entries.

Parameter name	Data type	Required?	Definition
			PublicKeyCredentialParameters' algorithm identifiers are values that SHOULD be registered in the IANA COSE Algorithms registry [IANA-COSE-ALGS-REG].
excludeList (0x05)	Array of `PublicKeyCredentialDescriptor`	Optional	An array of PublicKeyCredentialDescriptor structures, as specified in [WebAuthn]. The authenticator returns an error if the authenticator already contains one of the credentials enumerated in this array. This allows RPs to limit the creation of multiple credentials for the same account on a single authenticator. If this parameter is present, it MUST NOT be empty.
extensions (0x06)	CBOR map of extension identifier → authenticator extension input values	Optional	Parameters to influence authenticator operation, as specified in [WebAuthn]. These parameters might be authenticator specific.
options (0x07)	Map of authenticator options	Optional	Parameters to influence authenticator operation, as specified in in the table below.
pinUvAuthParam (0x08)	Byte String	Optional	Result of calling authenticate(pinUvAuthToken, clientDataHash)
pinUvAuthProtocol (0x09)	Unsigned Integer	Optional	PIN/UV protocol version chosen by the platform
enterpriseAttestation (0x0A)	Unsigned Integer	optional	An authenticator supporting this enterprise attestation feature is **enterprise attestation capable** and signals its support via the ep Option ID in the authenticatorGetInfo command response. If the `enterpriseAttestation` parameter is absent, attestation's privacy characteristics are unaffected, regardless of whether the enterprise attestation feature is presently enabled. If present with a valid value, the usual privacy concerns around attestation batching may not apply to the results of this operation and the platform is requesting an enterprise attestation that includes uniquely identifying information.
attestationFormatsPreference (0x0B)	Array of String	optional	A prioritized list of attestation statement format identifiers that the client and/or RP prefers. Authenticators that support multiple formats may use this list to select a format compatible with the caller. Clients may request omission of attestation by including a single element with the string value `"none"`.

The following option keys are defined for use in authenticatorMakeCredential's options parameter. All option keys have boolean values.

NOTE: For brevity, individual option keys are often referred to as simply an "option", below.

Option Key	Default value	Definition
rk	false	Specifies whether this credential is to be discoverable or not.
up	true	user presence: Instructs the authenticator to require user consent to complete the operation. Platforms MAY send the "up" option key to CTAP2.1 authenticators, and its value MUST be `true` if present. The value `false` will cause a CTAP2_ERR_INVALID_OPTION response regardless of authenticator version.
uv	false	user verification: If `true`, instructs the authenticator to require a user-verifying gesture in order to complete the request. Examples of such gestures are fingerprint scan or a PIN. NOTE: Use of this "uv" option key is deprecated in CTAP2.1. Instead, platforms SHOULD create a pinUvAuthParam by obtaining pinUvAuthToken via getPinUvAuthTokenUsingUvWithPermissions or getPinUvAuthTokenUsingPinWithPermissions, as appropriate. Platforms MUST NOT include the "uv" option key if the authenticator does not support built-in user verification. Platforms MUST NOT include both the "uv" option key and the pinUvAuthParam parameter in the same request.

NOTE: For backwards compatibility, platforms must be aware that if a FIDO_2_0 (aka CTAP2.0) authenticator is protected by some form of user verification, it always requires some form of user verification for authenticatorMakeCredential operations. If a platform attempts to create a non-discoverable credential on a CTAP2.0 authenticator without including the "uv" option key or the pinUvAuthToken parameter that authenticator will return an error. In contrast, a FIDO_2_1 (aka CTAP2.1) authenticator with the makeCredUvNotRqd option ID (set to `true`) in the authenticatorGetInfo response structure, will allow the creation of non-discoverable credentials without requiring some form of user verification.

NOTE: For backwards compatibility, platforms must be aware that FIDO_2_0 (aka CTAP2.0) authenticators will return a CTAP2_ERR_INVALID_OPTION response if "up" is present. Platforms SHOULD *NOT* send "up" to a CTAP2.0 authenticator.

NOTE: The [WebAuthn] specification defines an abstract `authenticatorMakeCredential` operation, which corresponds to the operation described in this section. The parameters in the abstract [WebAuthn] `authenticatorMakeCredential` operation map to the above parameters as follows:

[WebAuthn] `authenticatorMakeCredential` operation	**CTAP `authenticatorMakeCredential` operation**
hash	clientDataHash
rpEntity	rp
userEntity	user
requireResidentKey	options.rk
requireUserPresence	options.up NOTE: [WebAuthn-2] defines `requireUserPresence` as a constant Boolean value `true`. options.up is required to be absent for backwards comparability with CTAP2.0.
requireUserVerification	options.uv or pinUvAuthParam
credTypesAndPubKeyAlgs	pubKeyCredParams
excludeCredentialDescriptorList	excludeList
attestationFormats	attestationFormatsPreference
extensions	extensions

NOTE: Icon values used with authenticators can employ [RFC2397] "data" URLs so that the image data is passed by value, rather than by reference. This can enable authenticators with a display but no Internet connection to display icons.

NOTE: Text strings are UTF-8 encoded (CBOR major type 3).

6.1.1. Platform Actions for `authenticatorMakeCredential` (non-normative)

To invoke `authenticatorMakeCredential`, the platform performs the following steps, in general. Here, we are assuming that the platform has already queried the authenticator for its particulars using the authenticatorGetInfo command, and has determined that the authenticator's present characteristics are likely sufficient to be able to satisfy the request(s) the platform will send it. In other words, this is only a brief sketch of plausible platform behavior.

For example, if the authenticator is not protected by some form of user verification and user verification is required for the present usage scenario, e.g., the Relying Party set `options.authenticatorSelection.userVerification` to "required" in the WebAuthn API, then the platform recovers in some fashion out of scope of these actions.

1. The platform marshals the necessary and appropriate input parameters given the present usage scenario, and additionally:
    1. If the authenticator is protected by some form of user verification, or the Relying Party prefers enforcing user verification (e.g., by setting `options.authenticatorSelection.userVerification` to "required", or "preferred" in the WebAuthn API):
        1. If the platform has already created a pinUvAuthParam parameter during this overall scenario, it uses that along with the other marshalled input parameters to invoke the authenticator operation: either `authenticatorMakeCredential` or possibly **authenticatorGetAssertion**. For example, in some situations (e.g., with CTAP2 authenticators) when an "exclude list" was provided by the Relying Party, the platform may first invoke the `authenticatorGetAssertion` operation multiple times to "pre-flight" the "exclude list" (i.e., to determine if any of the exclude list's credential IDs are already present on the authenticator), prior to invoking `authenticatorMakeCredential` to create a new credential on this authenticator.
        2. Otherwise, the platform examines various option IDs in the authenticatorGetInfo response to determine its course of action:
            1. If the uv option ID is present and set to `true`:
                1. If the pinUvAuthToken option ID is present and `true` then plan to use getPinUvAuthTokenUsingUvWithPermissions to obtain a pinUvAuthToken, and let it be the *selected operation*. Go to Step 1.1.2.3.
                2. Else (implying the pinUvAuthToken option ID is set to `false` or absent) use the "uv" option key when invoking the `authenticatorMakeCredential` operation and terminate these steps. (Note that if the authenticator returns a 0x36 error code (CTAP2_ERR_PUAT_REQUIRED (aka CTAP2_ERR_PIN_REQUIRED in CTAP2.0)) then "fall back" and go to Step 1.1.2.2.2.1)
            2. Else (implying the uv option ID is present and set to `false` or absent):
                1. If the pinUvAuthToken option ID is present and `true`:
                    1. To continue, ensure the clientPin option ID is present and `true`. Plan to use getPinUvAuthTokenUsingPinWithPermissions to obtain a pinUvAuthToken, and let it be the *selected operation*. Go to Step 1.1.2.3.
                2. Else (implying the pinUvAuthToken option ID is absent):
                    1. To continue, ensure the clientPin option ID is present and `true`. Plan to use getPinToken to obtain a pinUvAuthToken, and let it be the *selected operation*.
            3. In preparation for obtaining pinUvAuthToken, the platform:
                1. Obtains a shared secret.
                2. Sets the pinUvAuthProtocol parameter to the value as selected when it obtained the shared secret.
            4. Then the platform obtains a pinUvAuthToken from the authenticator, with the mc (and likely also with the ga) permission (see "pre-flight", mentioned above), using the *selected operation*.
            5. If pinUvAuthToken was obtained successfully:
                1. The platform creates the pinUvAuthParam parameter by calling authenticate(pinUvAuthToken, clientDataHash), and goes to Step 1.1.1.
            6. Else (implying pinUvAuthToken was not obtained successfully):
                1. If the error code when attempting to obtain the pinUvAuthToken is one of the following: CTAP2_ERR_NOT_ALLOWED, CTAP2_ERR_UV_BLOCKED or

CTAP2_ERR_UNAUTHORIZED_PERMISSION, and the *selected operation* is getPinUvAuthTokenUsingUvWithPermissions:

1. The platform falls back to PIN authentication, and goes to Step 1.1.2.2.

2. Else:

1. Fails this overall scenario

2. Otherwise, implying the authenticator is not presently protected by some form of user verification, or the Relying Party wants to create a non-discoverable credential and not require user verification (e.g., by setting `options.authenticatorSelection.userVerification` to "discouraged" in the WebAuthn API), the platform invokes the `authenticatorMakeCredential` operation using the marshalled input parameters along with the 'uv' option key set to `false` and terminate these steps.

6.1.2. `authenticatorMakeCredential` Algorithm

Upon receipt of an `authenticatorMakeCredential` request, the authenticator performs the following procedure:

1. If authenticator supports either pinUvAuthToken or clientPin features and the platform sends a zero length pinUvAuthParam:

1. Request evidence of user interaction in an authenticator-specific way (e.g., flash the LED light).
2. If the user declines permission, or the operation times out, then end the operation by returning CTAP2_ERR_OPERATION_DENIED.
3. If evidence of user interaction is provided in this step then return either CTAP2_ERR_PIN_NOT_SET if PIN is not set or CTAP2_ERR_PIN_INVALID if PIN has been set.

> NOTE: This is done for backwards compatibility with CTAP2.0 platforms in the case where multiple authenticators are attached to the platform and the platform wants to enforce pinUvAuthToken feature semantics, but the user has to select which authenticator to get the pinUvAuthToken from. CTAP2.1 platforms SHOULD use § 6.9 authenticatorSelection (0x0B).

2. If the pinUvAuthParam parameter is present:

1. If the pinUvAuthProtocol parameter's value is not supported, return CTAP1_ERR_INVALID_PARAMETER error.
2. If the pinUvAuthProtocol parameter is absent, return CTAP2_ERR_MISSING_PARAMETER error.

3. Validate pubKeyCredParams with the following steps:

1. For each element of pubKeyCredParams:

1. If the element is missing required members, including members that are mandatory only for the specific `type`, then return an error, for example CTAP2_ERR_INVALID_CBOR.
2. If the values of any known members have the wrong type then return an error, for example CTAP2_ERR_CBOR_UNEXPECTED_TYPE.
3. If the element specifies an algorithm that is supported by the authenticator, and no algorithm has yet been chosen by this loop, then let the algorithm specified by the current element be the chosen algorithm.

2. If the loop completes and no algorithm was chosen then return CTAP2_ERR_UNSUPPORTED_ALGORITHM.

> NOTE: This loop chooses the first occurrence of an algorithm identifier supported by this authenticator but always iterates over every element of pubKeyCredParams to validate them.

4. Create a new authenticatorMakeCredential response structure and initialize both its "uv" bit and "up" bit as `false`.

5. If the options parameter is present, process all option keys and values present in the parameter. Treat any option keys that are not understood as absent.

> NOTE: As this specification defines normative behaviours for the 'rk', "up", and "uv" option keys, they MUST be understood by all authenticators.

1. If the "uv" option is absent, let the "uv" option be treated as being present with the value `false`. (This is the default)
2. If the pinUvAuthParam is present, let the "uv" option be treated as being present with the value `false`.

> NOTE: pinUvAuthParam and the "uv" option are processed as mutually exclusive with pinUvAuthParam taking precedence.

3. If the "uv" option is `true` then:

1. If the authenticator does not support a built-in user verification method end the operation by returning CTAP2_ERR_INVALID_OPTION.

2. If the built-in user verification method has not yet been enabled, end the operation by returning CTAP2_ERR_INVALID_OPTION.

4. If the "rk" option is present then:

    1. If the rk option ID is *not* present in authenticatorGetInfo response, end the operation by returning CTAP2_ERR_UNSUPPORTED_OPTION.

5. Else: (the "rk" option is absent)

    1. Let the "rk" option be treated as being present with the value `false`. (This is the default.)

6. If the "up" option is present then:

    1. If the "up" option is `false`, end the operation by returning CTAP2_ERR_INVALID_OPTION.

7. If the "up" option is absent, let the "up" option be treated as being present with the value `true` (i.e., this is the default for both CTAP2.0 and CTAP2.1 authenticators).

6. If the alwaysUv option ID is present and `true` then:

    1. Let the makeCredUvNotRqd option ID be treated as `false`.
    2. If the authenticator is *not* protected by some form of user verification:

        1. If the clientPin option ID is present and noMcGaPermissionsWithClientPin option ID is absent or `false` (clientPin is supported for the mc permission):

            1. End the operation by returning CTAP2_ERR_PUAT_REQUIRED.

        2. Else (clientPin is not supported):

            1. End the operation by returning CTAP2_ERR_OPERATION_DENIED.

    3. If the pinUvAuthParam is *not* present, and the uv option ID is `true`, let the "uv" option be treated as being present with the value `true`.

        > NOTE: The above step 6.3 is for backwards compatibility with CTAP2.0 platforms who are not aware of the Always UV feature.

    4. If the pinUvAuthParam is *not* present, and the "uv" option is `false` or absent:

        1. If the clientPin option ID is present and noMcGaPermissionsWithClientPin option ID is absent or `false` (clientPin is supported for the mc permission):

            1. End the operation by returning CTAP2_ERR_PUAT_REQUIRED.

        2. Else (clientPin is not supported):

            1. End the operation by returning CTAP2_ERR_OPERATION_DENIED.

7. If the makeCredUvNotRqd option ID is present and set to `true` in the authenticatorGetInfo response:

    1. If the following statements are all true:

        > NOTE: This step returns an error if the platform tries to create a discoverable credential without performing some form of user verification.

        1. The authenticator is protected by some form of user verification.
        2. The "uv" option is set to `false`.
        3. The pinUvAuthParam parameter is *not* present.
        4. The "rk" option is present and set to `true`.

        Then:

        1. If ClientPin option ID is `true` and the noMcGaPermissionsWithClientPin option ID is absent or `false`, end the operation by returning CTAP2_ERR_PUAT_REQUIRED.
        2. Otherwise, end the operation by returning CTAP2_ERR_OPERATION_DENIED.

8. Else: (the makeCredUvNotRqd option ID in authenticatorGetInfo's response is present with the value `false` or is absent):

    1. If the following statements are all true:

        > NOTE: This step returns an error if the platform tries to create a credential without performing some form of user verification when the makeCredUvNotRqd option ID in authenticatorGetInfo's response is present with the value `false` or is absent.

        1. The authenticator is protected by some form of user verification.
        2. The "uv" option is set to `false`.
        3. The pinUvAuthParam parameter is *not* present.

        Then:

1. If the ClientPin option ID is `true` and the noMcGaPermissionsWithClientPin option ID is absent or `false`, end the operation by returning CTAP2_ERR_PUAT_REQUIRED.
2. Otherwise, end the operation by returning CTAP2_ERR_OPERATION_DENIED.

9. If the enterpriseAttestation parameter is present:
   1. If the authenticator is *not* enterprise attestation capable, or the authenticator is enterprise attestation capable but enterprise attestation is disabled, then end the operation by returning CTAP1_ERR_INVALID_PARAMETER.
   2. Else: (the authenticator is enterprise attestation capable and enterprise attestation is enabled; see also § 7.1.2 Platform Actions):
      1. If the enterpriseAttestation parameter's value is *not* 1 or 2, then end the operation by returning CTAP2_ERR_INVALID_OPTION.
      2. Consider the following cases in order, until one matches, to learn whether the authenticator may return an enterprise attestation. (These substeps define when an authenticator is permitted to return an enterprise attestation. Authenticators MUST NOT do so in any other cases.)
         1. If the authenticator supports *only* vendor-facilitated enterprise attestation and the request's rp.id matches an entry on the authenticator's pre-configured RP ID list, then the authenticator MAY return an enterprise attestation.

            > NOTE: An authenticator that only supports vendor-facilitated enterprise attestation is obliged to treat enterpriseAttestation parameter values 1 and 2 equivalently, otherwise it will yield unexpected results if used with an enterprise-managed platform (which will be setting enterpriseAttestation to 2).

         2. If the authenticator supports vendor-facilitated enterprise attestation at all, the enterpriseAttestation parameter's value is 1, and the request's rp.id matches an entry on the authenticator's pre-configured RP ID list, then the authenticator MAY return an enterprise attestation.
         3. If the authenticator supports platform-managed enterprise attestation (whether or not vendor-facilitated enterprise attestation is also supported), and the enterpriseAttestation parameter's value is 2, then the platform MUST have performed the necessary vetting of the request's rp.id (e.g., via local policy lookup), and the authenticator MAY return an enterprise attestation without checking whether the request's rp.id matches an entry on the authenticator's pre-configured RP ID list (if any).
      3. If, by considering the substeps of the previous step, the authenticator did not conclude that it may return an enterprise attestation then let the enterpriseAttestation parameter be treated as absent, terminate these steps, and go to Step 10. A non-enterprise attestation will be returned with the credential.
      4. Apply any additional constraints that may prohibit returning an enterprise attestation. An authenticator has unlimited discretion to apply additional constraints which can further limit the contexts in which enterprise attestation is returned. They may be based on other parameters from the request or, indeed, on any other factor the authenticator wishes. It is the job of enterprise Relying Party to know the authenticators that it has deployed and thus to arrange the request so as to get its desired result.
      5. If, by considering any additional constraints in the previous step, the authenticator concluded that it did not wish to return an enterprise attestation then let the enterpriseAttestation parameter be treated as absent, terminate these steps, and go to Step 10. A non-enterprise attestation will be returned with the credential.
      6. If the authenticator has a display, then the authenticator SHOULD display an explicit warning to the user, including the rp.id, notifying the user that they are being uniquely identified to this Relying Party.
      7. Let epAtt in the authenticatorMakeCredential response structure be set to `true` and return an enterprise attestation.

10. If the following statements are all true:

    > NOTE: This step allows the authenticator to create a non-discoverable credential without requiring some form of user verification under the below specific criteria.

    1. "rk" and "uv" options are both set to `false` or omitted.
    2. the makeCredUvNotRqd option ID in authenticatorGetInfo's response is present with the value `true`.
    3. the pinUvAuthParam parameter is not present.

    Then go to Step 12.

    > NOTE: Step 4 has already ensured that the "uv" bit is `false` in the response.

11. If the authenticator is protected by some form of user verification, then:
    1. If pinUvAuthParam parameter is present (implying the "uv" option is `false` (see Step 5)):

1. Call verify(pinUvAuthToken, clientDataHash, pinUvAuthParam).
    1. If the verification returns `error`, then end the operation by returning CTAP2_ERR_PIN_AUTH_INVALID error.
2. Verify that the pinUvAuthToken has the mc permission, if not, then end the operation by returning CTAP2_ERR_PIN_AUTH_INVALID.
3. If the pinUvAuthToken has a permissions RP ID associated:
    1. If the permissions RP ID does not match the rp.id in this request, then end the operation by returning CTAP2_ERR_PIN_AUTH_INVALID.
4. Let userVerifiedFlagValue be the result of callinggetUserVerifiedFlagValue().
5. If userVerifiedFlagValue is `false` then end the operation by returning CTAP2_ERR_PIN_AUTH_INVALID.
6. If userVerifiedFlagValue is `true` then set the "uv" bit to`true` in the response.
7. If the pinUvAuthToken does not have a permissions RP ID associated:
    1. Associate the request's rp.id parameter value with thepinUvAuthToken as its permissions RP ID.
8. Go to Step 12.

2. If the "uv" option is present and set to`true` (implying the pinUvAuthParam parameter is not present, and that the authenticator supports an enabled built-in user verification method, see Step 5):

    NOTE: This step provides backwards compatibility for CTAP2.0 platforms.

    1. Let internalRetry be `true`.
    2. Let uvState be the result of callingperformBuiltInUv(internalRetry)
    3. If uvState is `error`:
        1. If the error reason is a user action timeout, then return CTAP2_ERR_USER_ACTION_TIMEOUT.
        2. If the ClientPin option ID is `true` and the noMcGaPermissionsWithClientPin option ID is absent or `false`, end the operation by returning CTAP2_ERR_PUAT_REQUIRED.
        3. If the uvRetries counter is 0, return CTAP2_ERR_PIN_BLOCKED.
        4. Otherwise, end the operation by returning CTAP2_ERR_OPERATION_DENIED.
    4. If uvState is `success`:
        1. Set the "uv" bit to`true` in the response.

    NOTE: If Step 11 was skipped, then the authenticator is*NOT* protected by some form of user verification, and Step 4 has already ensured that the "uv" bit is`false` in the response.

12. If the excludeList parameter is present and contains a credential ID created by this authenticator, that is bound to the specified rp.id:
    1. If the credential's credProtect value is *not* userVerificationRequired, then:
        1. Let userPresentFlagValue be `false`.
        2. If the pinUvAuthParam parameter is present then let userPresentFlagValue be the result of calling getUserPresentFlagValue().
        3. Else, if evidence of user interaction was provided as part of Step 11 let userPresentFlagValue be `true`.
        4. If userPresentFlagValue is `false`, then:
            1. Wait for user presence.
            2. Regardless of whether user presence is obtained or the authenticator times out, terminate this procedure and return CTAP2_ERR_CREDENTIAL_EXCLUDED.
        5. Else, (implying userPresentFlagValue is `true`) terminate this procedure and return CTAP2_ERR_CREDENTIAL_EXCLUDED.

            NOTE: A user presence test is required for CTAP2 authenticators, before the RP is told that the authenticator is already registered, to behave similarly to CTAP1/U2F authenticators.
    2. Else (implying the credential's credProtect value is userVerificationRequired):
        1. If the "uv" bit is`true` in the response:
            1. Let userPresentFlagValue be `false`.
            2. If the pinUvAuthParam parameter is present then let userPresentFlagValue be the result of calling getUserPresentFlagValue().

3. Else, if evidence of user interaction was provided as part of Step 11 let userPresentFlagValue be `true`.
4. If userPresentFlagValue is `false`, then:
    1. Wait for user presence.
    2. Regardless of whether user presence is obtained or the authenticator times out, terminate this procedure and return CTAP2_ERR_CREDENTIAL_EXCLUDED.
5. Else, (implying userPresentFlagValue is `true`) terminate this procedure and return CTAP2_ERR_CREDENTIAL_EXCLUDED.

2. Else (implying user verification was not collected in Step 11), remove the credential from the excludeList and continue parsing the rest of the list.

13. If evidence of user interaction was provided as part of Step 11 (i.e., by invoking performBuiltInUv()):

> NOTE: This step's criteria implies that the "uv" option is present and set to `true` and the pinUvAuthParam parameter is not present. I.e., the pinUvAuthToken feature is not in use.

1. Set the "up" bit to `true` in the response.
2. Go to Step 15

14. If the "up" option is set to `true`:
    1. If the pinUvAuthParam parameter is present then:
        1. Let userPresentFlagValue be the result of calling getUserPresentFlagValue().
        2. If userPresentFlagValue is `false`:

> NOTE: An authenticator may be configured to collect user presence whenever the "up" option is `true` by setting the default user present time limit to zero.

            1. Request evidence of user interaction in an authenticator-specific way (e.g., flash the LED light). If the authenticator has a display, show the items contained within the user and rp parameter structures to the user, and request permission to create a credential.
            2. If the user declines permission, or the operation times out, then end the operation by returning CTAP2_ERR_OPERATION_DENIED.
    2. Else (implying the pinUvAuthParam parameter is not present):
        1. If the "up" bit is `false` in the response :
            1. Request evidence of user interaction in an authenticator-specific way (e.g., flash the LED light). If the authenticator has a display, show the items contained within the user and rp parameter structures to the user, and request permission to create a credential.
            2. If the user declines permission, or the operation times out, then end the operation by returning CTAP2_ERR_OPERATION_DENIED.
    3. Set the "up" bit to `true` in the response.
    4. Call clearUserPresentFlag(), clearUserVerifiedFlag(), and clearPinUvAuthTokenPermissionsExceptLbw().

> NOTE: This *consumes* both the "user present state", sometimes referred to as the "cached UP", and the "user verified state", sometimes referred to as "cached UV". These functions are no-ops if there is not an in-use pinUvAuthToken.

15. If the extensions parameter is present:
    1. Process any extensions that this authenticator supports, ignoring any that it does not support.
    2. Authenticator extension outputs generated by the authenticator extension processing are returned in the authenticator data. The set of keys in the authenticator extension outputs map MUST be equal to, or a subset of, the keys of the authenticator extension inputs map.

> NOTE: Some extensions may produce different output depending on the state of the "uv" bit and/or "up" bit in the response.

16. Generate a new credential key pair for the algorithm chosen in step 3.
17. If the "rk" option is set to `true`:
    1. The authenticator MUST create a discoverable credential.
    2. If a credential for the same rp.id and account ID already exists on the authenticator:
        1. If the existing credential contains a largeBlobKey, an authenticator MAY erase any associated large-blob data. Platforms MUST NOT assume that authenticators will do this. Platforms can later garbage collect any orphaned large-blobs.
        2. Overwrite that credential.
    3. Store the user parameter along with the newly-created key pair.

4. If authenticator does not have enough internal storage to persist the new credential, return CTAP2_ERR_KEY_STORE_FULL.

18. Otherwise, if the "rk" option is `false`: the authenticator MUST create a non-discoverable credential.

> NOTE: This step is a change from CTAP2.0 where if the "rk" option is `false` the authenticator could optionally create a discoverable credential.

19. If the authenticator doesn't support multiple attestation formats or the attestationFormatsPreference is absent or its value is the empty list, generate an attestation statement for the newly-created credential using clientDataHash, taking into account the value of the enterpriseAttestation parameter, if present, as described above in Step 9.

    If attestationFormatsPreference is present and contains only one entry with the value `"none"`, omit attestation from the output.

    If the authenticator supports multiple attestation formats and the attestationFormatsPreference parameter is present, the authenticator MUST choose a supported format whose attestation statement format identifier appears with the lowest index in the supplied array. If no supported format identifier appears on the list, the authenticator may select a format by any other means.

On success, the authenticator returns the following **authenticatorMakeCredential response structure** which contains an attestation object plus additional information.

Member name	Data type	Required?	Definition
fmt (0x01)	String	Required	The attestation statement format identifier.
authData (0x02)	Byte String	Required	The authenticator data object.
attStmt (0x03)	CBOR Map, the structure of which depends on the attestation statement format identifier	Optional	The attestation statement, as specified in [WebAuthn], if one is provided.
epAtt (0x04)	Boolean	Optional	Indicates whether an enterprise attestation was returned for this credential. If `epAtt` is absent or present and set to `false`, then an enterprise attestation was not returned. If `epAtt` is present and set to `true`, then an enterprise attestation was returned.
largeBlobKey (0x05)	Byte string	Optional	Contains the largeBlobKey for the credential, if requested with the largeBlobKey extension.
unsignedExtensionOutputs (0x06)	CBOR map of extension identifier → unsigned extension output values	Optional	A map, keyed by extension identifiers, to unsigned outputs of extensions, if any. Authenticators SHOULD omit this field if no processed extensions define unsigned outputs. Clients MUST treat an empty map the same as an omitted field.

6.1.3. Discoverable credentials

A credential may, or may not, be *discoverable*. A discoverable credential [WebAuthn] has the property that, in response to an authenticatorGetAssertion request where the `allowList` parameter is omitted, the authenticator is able to *discover* the appropriate public key credential source given only an RP ID, possibly with user assistance.

Each credential has a credential protection policy. For backwards compatibility with CTAP2.0 platforms, the default credential creation policy is userVerificationOptional (0x01). If a credential was created with credential protection values of userVerificationOptionalWithCredentialIDList (0x02) or userVerificationRequired (0x03) it will not be discoverable unless the platform invokes authenticatorGetAssertion with a valid pinUvAuthParam or the "uv" option key with a value of `true`.

> NOTE: Regarding user assistance, for example, the authenticator may provide the user a pick-list of credentials scoped to the RP ID.

In contrast, server-side credentials (also known as **non-discoverable credentials**) have the property that their

credential IDs MUST be supplied by the Relying Party in authenticatorGetAssertion's `allowList` parameter in order for the authenticator to discover and employ them.

Note that this definition does not speak to whether a credential is *statefully maintained* or not.

An authenticator may choose to keep state, such as the private key, whether a credential is discoverable or not (see also public key credential source). A discoverable credential, however, always involves maintaining some state because it must be discoverable using only the RP ID and the `user id` (also known as the user handle) must always be returned.

All state that is kept for a discoverable credential MUST be stored client side—i.e., such that the authenticator working together with the client platform, if necessary, can satisfy requested authenticator operations.

An authenticator specifies whether it is capable of creating discoverable credentials via the rk option ID in the authenticatorGetInfo response. A discoverable credential will be created if, and only if, the rk option key of the options parameter of an authenticatorMakeCredential request is `true`.

If the authenticatorCredentialManagement command is supported by an authenticator then it can be used to manage discoverable credentials.

If a discoverable credential's state is deleted, e.g., by the authenticatorCredentialManagement command or overwritten by authenticatorMakeCredential, the associated credentialID MUST no longer yield a public key credential source, e.g., when processed by the authenticator's equivalent of the Lookup Credential Source by Credential ID Algorithm including cases where the credential source is encoded within the credentialID. This means, for example, that any such deleted credentials whose credentialIDs may have been stored server-side and subsequently are provided in an allowList to authenticatorGetAssertion, will no longer be "located" in the latter's Step 7 when the allowList is processed.

> NOTE: Historically discoverable credentials have been called "resident keys", and this terminology can still be found in aspects of the protocol. (For example the name of the rk option key comes from the term "resident key".) However, the word "resident" conflated the concepts of being discoverable and being statefully maintained by the authenticator, when it's only the former that is externally observable and thus important.

6.2. authenticatorGetAssertion (0x02)

This method is used by a host to request cryptographic proof of user authentication as well as user consent to a given transaction, using a previously generated credential that is bound to the authenticator and relying party identifier. It takes the following **input parameters**, several of which correspond to those defined in the authenticatorGetAssertion operation section of the Web Authentication specification:

Parameter name	Data type	Required?	Definition
rpId (0x01)	String	Required	relying party identifier. See [WebAuthn].
clientDataHash (0x02)	Byte String	Required	Hash of the serialized client data collected by the host. See [WebAuthn].
allowList (0x03)	Array of `PublicKeyCredentialDescriptor`	Optional	An array of PublicKeyCredentialDescriptor structures, each denoting a credential, as specified in [WebAuthn]. A platform MUST NOT send an empty allowList—if it would be empty it MUST be omitted. If this parameter is present the authenticator MUST only generate an assertion using one of the denoted credentials.
extensions (0x04)	CBOR map of extension identifier → authenticator extension input values	Optional	Parameters to influence authenticator operation. These parameters might be authenticator specific.
options (0x05)	Map of authenticator options	Optional	Parameters to influence authenticator operation, as specified in the table below.
pinUvAuthParam (0x06)	Byte String	Optional	Result of calling authenticate(pinUvAuthToken, clientDataHash)

Parameter name	Data type	Required?	Definition
pinUvAuthProtocol (0x07)	Unsigned Integer	Optional	PIN/UV protocol version selected by platform.

The following option keys are defined for use in authenticatorGetAssertion's options parameter. All option keys have boolean values.

NOTE: For brevity, individual option keys are often referred to as simply an "bption", below.

Option Key	Default value	Definition
up	true	user presence: Instructs the authenticator to require user consent to complete the operation.
uv	false	user verification: If `true`, instructs the authenticator to require a user-verifying gesture in order to complete the request. Examples of such gestures are fingerprint scan or a PIN. NOTE: Use of this "uv" option key is deprecated in CTAP2.1. Instead, platforms SHOULD create a pinUvAuthParam by obtaining pinUvAuthToken via getPinUvAuthTokenUsingUvWithPermissions or getPinUvAuthTokenUsingPinWithPermissions, as appropriate. Platforms MUST NOT include the "uv" option parameter if the authenticator does not support built-in user verification. Platforms MUST NOT include both "uv" and pinUvAuthParam parameters in same request.

NOTE: Platforms MUST NOT send the "**rk**" option key.

NOTE: For backwards compatibility with CTAP2.0 platforms, the authenticator MAY perform a built-in user verification method even if not requested to enhance its security offering. Thus, platforms SHOULD be prepared to receive a CTAP2_ERR_PUAT_REQUIRED error even if the platform did not include the "uv" option key, or did include it and set it to `false`. CTAP2.1 authenticators SHOULD use the authenticator always requires some form of user verification feature to signal this behaviour.

NOTE: The [WebAuthn] specification defines an abstract `authenticatorGetAssertion` operation, which corresponds to the operation described in this section. The parameters in the abstract [WebAuthn] `authenticatorGetAssertion` operation map to the above parameters as follows:

[WebAuthn] **`authenticatorGetAssertion` operation**	**CTAP `authenticatorGetAssertion` operation**
hash	clientDataHash
rpId	rpId
allowCredentialDescriptorList	allowList
requireUserPresence	options.up NOTE: [WebAuthn-2] defines `requireUserPresence` as a constant Boolean value `true`. options.up may be set to `false` in CTAP "pre-flight" commands but is always set to `true` for any `authenticatorGetAssertion` request that is intended to generate an assertion that will be returned to an Relying Party via the WebAuthn API. This is because such an assertion must have the "user present" bit of the "flags bits" of the authenticator data set to `true` to be considered valid by clients of the WebAuthn API.
requireUserVerification	options.uv or pinUvAuthParam
extensions	extensions

6.2.1. Platform Actions for `authenticatorGetAssertion` (non-normative)

To invoke `authenticatorGetAssertion`, the platform performs the following steps, in general. Here, we are assuming that the platform has already queried the authenticator for its particulars using the authenticatorGetInfo command, and has determined that the authenticator's present characteristics are likely sufficient to be able to

satisfy the request(s) the platform will send it. In other words, this is only a brief sketch of plausible platform behavior.

For example, if the authenticator is not protected by some form of user verification and user verification is required for the present usage scenario, e.g., the Relying Party set `options.userVerification` to "required" in the WebAuthn API, then the platform recovers in some fashion out of scope of these actions.

1. The platform marshals the necessary and appropriate input parameters given the present usage scenario, and additionally:
    1. If the authenticator is protected by some form of user verification or the Relying Party prefers enforcing user verification (e.g., by setting `options.userVerification` to "required", or "preferred" in the WebAuthn API):
        1. If the platform has already created a pinUvAuthParam parameter during this overall scenario, it uses that along with the other marshalled input parameters to invoke the `authenticatorGetAssertion`. Or, in some situations (e.g., with CTAP2 authenticators) the platform may invoke the `authenticatorGetAssertion` operation multiple times using the pinUvAuthParam parameter to "pre-flight" an "allow list" (i.e., to determine if any of the allow list's credential IDs are already present on the authenticator), prior to invoking `authenticatorGetAssertion` to have this authenticator issue an assertion using the selected credential.
        2. Otherwise, the platform examines various option IDs in the authenticatorGetInfo response to determine its course of action:
            1. If the uv option ID is present and set to `true`:
                1. If the pinUvAuthToken option ID is present and `true` then plan to use getPinUvAuthTokenUsingUvWithPermissions to obtain a pinUvAuthToken, and let it be the *selected operation*. Go to Step 1.1.2.3.
                2. Else (implying the pinUvAuthToken option ID is set to `false` or absent) use the 'uv' option key when invoking the `authenticatorGetAssertion` operation and terminate these steps. (Note that if the authenticator returns a 0x36 error code (CTAP2_ERR_PUAT_REQUIRED (aka CTAP2_ERR_PIN_REQUIRED in CTAP2.0)) then "fall back" and go to Step 1.1.2.2.2.1)
            2. Else (implying the uv option ID is present and set to `false` or absent):
                1. If the pinUvAuthToken option ID is present and `true`:
                    1. To continue, ensure the clientPin option ID is present and `true`. Plan to use getPinUvAuthTokenUsingPinWithPermissions to obtain a pinUvAuthToken, and let it be the *selected operation*. Go to Step 1.1.2.3.
                2. Else (implying the pinUvAuthToken option ID is absent):
                    1. To continue, ensure the clientPin option ID is present and `true`. Plan to use getPinToken to obtain a pinUvAuthToken, and let it be the *selected operation*.
            3. In preparation for obtaining pinUvAuthToken, the platform:
                1. Obtains a shared secret.
                2. Sets the pinUvAuthProtocol parameter to the value as selected when it obtained the shared secret.
            4. Then the platform obtains a pinUvAuthToken from the authenticator, with the ga permission using the *selected operation*.
            5. If pinUvAuthToken was obtained successfully:
                1. The platform creates the pinUvAuthParam parameter by calling authenticate(pinUvAuthToken, clientDataHash), and goes to Step 1.1.1 to use it.
            6. Else (implying pinUvAuthToken was not obtained successfully):
                1. If the error code when attempting to obtain the pinUvAuthToken is one of the following: CTAP2_ERR_NOT_ALLOWED, CTAP2_ERR_UV_BLOCKED or CTAP2_ERR_UNAUTHORIZED_PERMISSION, and the *selected operation* is getPinUvAuthTokenUsingUvWithPermissions:
                    1. The platform falls back to PIN authentication, and goes to Step 1.1.2.2.1.
                2. Else:
                    1. Fails this overall scenario
    2. Otherwise, implying the authenticator is not presently protected by some form of user verification, or the Relying Party does not wish to require user verification (e.g., by setting `options.userVerification` to "discouraged" in the WebAuthn API), the platform invokes the `authenticatorGetAssertion` operation using the marshalled input parameters along with an absent 'uv' option key.

6.2.2. `authenticatorGetAssertion` Algorithm

Upon receipt of a `authenticatorGetAssertion` request, the authenticator performs the following procedure:

1. If authenticator supports either pinUvAuthToken or clientPin features and the platform sends a zero length pinUvAuthParam:
    1. Request evidence of user interaction in an authenticator-specific way (e.g., flash the LED light).
    2. If the user declines permission, or the operation times out, then end the operation by returning CTAP2_ERR_OPERATION_DENIED.
    3. If evidence of user interaction is provided in this step then return either CTAP2_ERR_PIN_NOT_SET if PIN is not set or CTAP2_ERR_PIN_INVALID if PIN has been set.

    > NOTE: This is done for backwards compatibility with CTAP2.0 platforms in the case where multiple authenticators are attached to the platform and the platform wants to enforce pinUvAuthToken semantics, but the user has to select which authenticator to get the pinUvAuthToken from. CTAP2.1 platforms SHOULD use § 6.9 authenticatorSelection (0x0B).

2. If the pinUvAuthParam parameter is present:
    1. If the pinUvAuthProtocol parameter's value is not supported, return CTAP1_ERR_INVALID_PARAMETER error.
    2. If the pinUvAuthProtocol parameter is absent, return CTAP2_ERR_MISSING_PARAMETER error.
3. Create a new authenticatorGetAssertion response structure and initialize both its "uv" bit and "up" bit as `false`.
4. If the options parameter is present, process all option keys and values present in the parameter. Treat any option keys that are not understood as absent.

    > NOTE: As this specification defines normative behaviours for the "rk", "up", and "uv" option keys, they MUST be understood by all authenticators.

    1. If the "uv" option is absent, let the "uv" option be treated as being present with the value `false`. (This is the default)
    2. If the pinUvAuthParam is present, let the "uv" option be treated as being present with the value `false`.

        > NOTE: pinUvAuthParam and the "uv" option are processed as mutually exclusive with pinUvAuthParam taking precedence.

    3. If the "uv" option is present and `true` then:
        1. If the authenticator does not support a built-in user verification method end the operation by returning CTAP2_ERR_INVALID_OPTION.
        2. If the built-in user verification method has not yet been enabled, end the operation by returning CTAP2_ERR_INVALID_OPTION.
    4. If the "rk" option is present then:
        1. Return CTAP2_ERR_UNSUPPORTED_OPTION.
    5. If the "up" option is *not* present then:
        1. Let the "up" option be treated as being present with the value `true`. (This is the default)
5. If the alwaysUv option ID is present and `true` and the "up" option is present and `true` then:
    1. If the authenticator is *not* protected by some form of user verification:
        1. If the clientPin option ID is present and noMcGaPermissionsWithClientPin option ID is absent or `false` (clientPin is supported for the ga permission):
            1. End the operation by returning CTAP2_ERR_PUAT_REQUIRED.
        2. Else (clientPin is not supported):
            1. End the operation by returning CTAP2_ERR_OPERATION_DENIED.
    2. If the pinUvAuthParam is present then go to Step 6.
    3. If the "uv" option is `true` then go to Step 6.
    4. If the "uv" option is `false` and the authenticator supports a built-in user verification method, and the user verification method is enabled then:
        1. Let the "uv" option be treated as being present with the value `true`.
        2. Go To Step 6.
    5. If the clientPin option ID is present and noMcGaPermissionsWithClientPin option ID is absent or `false`, then:

        > NOTE: This is to address the case of CTAP2.0 platforms not being aware of and ignoring the alwaysUv option ID.

1. End the operation by returning CTAP2_ERR_PUAT_REQUIRED.

6. Else (clientPin is not supported):

    1. End the operation by returning CTAP2_ERR_OPERATION_DENIED.

6. If authenticator is protected by some form of user verification, then:

    1. If pinUvAuthParam parameter is present (implying the "uv" option is treated as `false`, see Step 4):

        1. Call verify(pinUvAuthToken, clientDataHash, pinUvAuthParam).

            1. If the verification returns `error`, return CTAP2_ERR_PIN_AUTH_INVALID error.
            2. If the verification returns `success`, set the "uv" bit to `true` in the response.

        2. Let userVerifiedFlagValue be the result of calling getUserVerifiedFlagValue().
        3. If userVerifiedFlagValue is `false` then end the operation by returning CTAP2_ERR_PIN_AUTH_INVALID.
        4. Verify that the pinUvAuthToken has the ga permission, if not, return CTAP2_ERR_PIN_AUTH_INVALID.
        5. If the pinUvAuthToken has a permissions RP ID associated:

            1. If the permissions RP ID does not match the `rpId` in this request, return CTAP2_ERR_PIN_AUTH_INVALID.

        6. If the pinUvAuthToken does not have a permissions RP ID associated:

            1. Associate the request's `rpId` parameter value with the pinUvAuthToken as its permissions RP ID.

        7. Go to Step 7.

    2. If the "uv" option is present and set to `true` (implying the pinUvAuthParam parameter is not present, and that the authenticator supports an enabled built-in user verification method, see Step 4):

        > NOTE: This step provides backwards compatibility for CTAP2.0 platforms.

        1. Let internalRetry be `true`.
        2. Let uvState be the result of calling performBuiltInUv(internalRetry)
        3. If uvState is `error`:

            1. If the error reason is a user action timeout, then return CTAP2_ERR_USER_ACTION_TIMEOUT.
            2. If the ClientPin option ID is `true` and the noMcGaPermissionsWithClientPin option ID is absent or `false`, end the operation by returning CTAP2_ERR_PUAT_REQUIRED.
            3. If the uvRetries counter is 0, return CTAP2_ERR_PIN_BLOCKED.
            4. Otherwise, end the operation by returning CTAP2_ERR_OPERATION_DENIED.

        4. If uvState is success:

            1. Set the "uv" bit to `true` in the response.

    > NOTE: If Step 6 was skipped, then the authenticator is *NOT* protected by some form of user verification, and Step 3 has already ensured that the "uv" bit is `false` in the response.

7. Locate all credentials that are eligible for retrieval under the specified criteria:

    1. If the allowList parameter is present and is non-empty, locate all denoted credentials created by this authenticator and bound to the specified `rpId`.
    2. If an allowList is not present, locate all discoverable credentials that are created by this authenticator and bound to the specified `rpId`.
    3. Create an **applicable credentials list** populated with the located credentials.
    4. Iterate through the applicable credentials list, and if credential protection for a credential is marked as `userVerificationRequired`, and the "uv" bit is `false` in the response, remove that credential from the applicable credentials list.
    5. Iterate through the applicable credentials list, and if credential protection for a credential is marked as `userVerificationOptionalWithCredentialIDList` and there is no allowList passed by the client and the "uv" bit is `false` in the response, remove that credential from the applicable credentials list.
    6. If the applicable credentials list is empty, return CTAP2_ERR_NO_CREDENTIALS.
    7. Let `numberOfCredentials` be the number of applicable credentials found.

8. If evidence of user interaction was provided as part of Step 6.2 (i.e., by invoking performBuiltInUv()):

    > NOTE: This step's criteria implies that the "uv" option is present and set to `true` and the pinUvAuthParam parameter is not present. I.e., the pinUvAuthToken feature is not in use.

    1. Set the "up" bit to `true` in the response.

2. Go to Step 10

9. If the "up" option is set to `true` or not present:
    1. If the pinUvAuthParam parameter is present then:
        1. Let userPresentFlagValue be the result of callinggetUserPresentFlagValue().
        2. If userPresentFlagValue is `false`:

            NOTE: An authenticator may be configured to collect user presence whenever the "up" option is `true` by setting the default user present time limit to zero.

            1. Request evidence of user interaction in an authenticator-specific way (e.g., flash the LED light). If the authenticator has a display, show the `rpId` parameter value to the user, and request permission to create an assertion.
            2. If the user declines permission, or the operation times out, then end the operation by returning CTAP2_ERR_OPERATION_DENIED.
    2. Else (implying the pinUvAuthParam parameter is not present):
        1. If the "up" bit is `false` in the response:
            1. Request evidence of user interaction in an authenticator-specific way (e.g., flash the LED light). If the authenticator has a display, show the `rpId` parameter value to the user, and request permission to create an assertion.
            2. If the user declines permission, or the operation times out, then end the operation by returning CTAP2_ERR_OPERATION_DENIED.
    3. Set the "up" bit to `true` in the response.
    4. Call clearUserPresentFlag(), clearUserVerifiedFlag(), and clearPinUvAuthTokenPermissionsExceptLbw().

        NOTE: This *consumes* both the "user present state", sometimes referred to as the "cached UP", and the "user verified state", sometimes referred to as "cached UV". These functions are no-ops if there is not an in-use pinUvAuthToken.

10. If the extensions parameter is present:
    1. Process any extensions that this authenticator supports, ignoring any that it does not support.
    2. Authenticator extension outputs generated by the authenticator extension processing are returned in the authenticator data. The set of keys in theauthenticator extension outputs map MUST be equal to, or a subset of, the keys of the authenticator extension inputs map.

    NOTE: Some extensions may produce different output depending on the state of the "uv" and/or "up" bits set in the response.

11. If the allowList parameter is present:
    1. Select any credential from the applicable credentials list.
    2. Delete the `numberOfCredentials` member.
    3. Go to Step 13.
12. If allowList is not present:
    1. If `numberOfCredentials` is one:
        1. Select that credential.
    2. If `numberOfCredentials` is more than one:
        1. Order the credentials in the applicable credentials list by the time when they were created in reverse order. (I.e. the first credential is the most recently created.)
        2. If the authenticator does not have a display, or the authenticator does have a display and the "uv" and "up" options are `false`:
            1. Remember the authenticatorGetAssertion parameters.
            2. Create a credential counter (`credentialCounter`) and set it to 1. This counter signifies the next credential to be returned by the authenticator, assuming zero-based indexing.
            3. Start a timer. This is used during authenticatorGetNextAssertion command. This step is OPTIONAL if transport is done over NFC.
            4. Select the first credential.
        3. If authenticator has a display and at least one of the "uv" and "up" options is `true`:
            1. Display all the credentials in theapplicable credentials list to the user, using their friendly name along with other stored account information.

2. Also, display the `rpId` of the requester (specified in the request) and ask the user to select a credential.
3. If the user declines to select a credential or takes too long (as determined by the authenticator), terminate this procedure and return the CTAP2_ERR_OPERATION_DENIED error.
4. Update the response to set the `userSelected` member to `true` and to delete the `numberOfCredentials` member.
5. Select the credential indicated by the user.

3. Update the response to include the selected credential's publicKeyCredentialUserEntity information. User identifiable information (name, DisplayName, icon) inside the publicKeyCredentialUserEntity MUST NOT be returned if user verification is not done by the authenticator.

13. Sign the clientDataHash along with authData with the selected credential, using the structure specified in [WebAuthn].

On success, the authenticator returns the following **authenticatorGetAssertion response structure**:

Member name	Data type	Required?	Definition
credential (0x01)	`PublicKeyCredentialDescriptor`	Required	PublicKeyCredentialDescriptor structure containing the credential identifier whose private key was used to generate the assertion.
authData (0x02)	Byte String	Required	The signed-over contextual bindings made by the authenticator, as specified in [WebAuthn].
signature (0x03)	Byte String	Required	The assertion signature produced by the authenticator, as specified in [WebAuthn].
user (0x04)	`PublicKeyCredentialUserEntity`	Optional	PublicKeyCredentialUserEntity structure containing the user account information. User identifiable information (name, DisplayName, icon) MUST NOT be returned if user verification is not done by the authenticator. U2F Devices: For U2F devices, this parameter is not returned as this user information is not present for U2F credentials. FIDO Devices - server-side credentials: For server-side credentials on FIDO devices, this parameter is OPTIONAL as server-side credentials behave the same as U2F credentials where they are discovered given the user information on the RP. Authenticators MAY store user information inside the credential ID. FIDO Devices - discoverable credentials: For discoverable credentials on FIDO devices, at least `user` "`id`" is mandatory. For single account per RP case, authenticator returns "`id`" field to the platform which will be returned to the [WebAuthn] layer. For multiple accounts per RP case, where the authenticator does not have a display, authenticator returns "id" as well as other fields to the platform. Platform will use this information to show the account selection UX

Member name	Data type	Required?	Definition
			to the user and for the user selected account, it will ONLY return "id" back to the [WebAuthn] layer and discard other user details.
numberOfCredentials (0x05)	Integer	Optional	Total number of account credentials for the RP. Optional; defaults to one. This member is required when more than one credential is found for an RP, and the authenticator does not have a display or the UV & UP flags are false. Omitted when returned for the authenticatorGetNextAssertion method.
userSelected (0x06)	Boolean	Optional	Indicates that a credential was selected by the user via interaction directly with the authenticator, and thus the platform does not need to confirm the credential. Optional; defaults to `false`. MUST NOT be present in response to a request where an `allowList` was given, where `numberOfCredentials` is greater than one, nor in response to an `authenticatorGetNextAssertion` request.
largeBlobKey (0x07)	Byte string	Optional	The contents of the associated largeBlobKey if present for the asserted credential, and if `largeBlobKey` was true in the extensions input.
unsignedExtensionOutputs (0x08)	CBOR map of extension identifier → unsigned extension output values	Optional	A map, keyed by extension identifiers, to unsigned outputs of extensions, if any. Authenticators SHOULD omit this field if no processed extensions define unsigned outputs. Clients MUST treat an empty map the same as an omitted field.

Within the "flags bits" of the authenticator data structure returned, the authenticator will report what was actually done within the authenticator boundary. The meanings of the combinations of the User Present (UP) and User Verified (UV) bit flags are as follows:

Flags	Meaning
"up"=0 "uv"=0	Silent authentication
"up"=1 "uv"=0	Physical user presence verified, but no user verification
"up"=0 "uv"=1	User verification performed, but physical user presence not verified. NOTE: Returning an assertion with the "up" bit set to `false` is not considered valid at the WebAuthn API layer [WebAuthn-2], and typically is only used for "pre-flighting".
"up"=1 "uv"=1	User verification performed and physical user presence verified

6.3. authenticatorGetNextAssertion (0x08)§

The client calls this method when the authenticatorGetAssertion response contains the numberOfCredentials member and the number of credentials exceeds 1. This method is used to obtain the next per-credential signature for a given authenticatorGetAssertion request. It takes no arguments.

NOTE: this is a stateful command and the specified implementation accommodations apply to it.

When this command is received, the authenticator performs the following procedure:

1. If authenticator does not remember any authenticatorGetAssertion parameters, return CTAP2_ERR_NOT_ALLOWED.
2. If the `credentialCounter` is equal to or greater than `numberOfCredentials`, return CTAP2_ERR_NOT_ALLOWED.
3. If timer since the last call to authenticatorGetAssertion/authenticatorGetNextAssertion is greater than 30 seconds, discard the current authenticatorGetAssertion state and return CTAP2_ERR_NOT_ALLOWED. This step is OPTIONAL if transport is done over NFC.

   > NOTE: the section on stateful commands makes this timeout OPTIONAL for any stateful command. This section supersedes that and makes it mandatory in this instance, except over NFC, where maintaining timers for that length of time can be problematic.

4. Select the credential indexed by `credentialCounter`. (I.e. credentials[n] assuming a zero-based array.)
5. Update the response to include the selected credential's publicKeyCredentialUserEntity information. User identifiable information (name, DisplayName, icon) inside the publicKeyCredentialUserEntity MUST NOT be returned if user verification was not done by the authenticator in the original authenticatorGetAssertion call.
6. Sign the clientDataHash along with authData with the selected credential, using the structure specified in [WebAuthn].
7. Reset the timer. This step is OPTIONAL if transport is done over NFC.
8. Increment `credentialCounter`.

On success, the authenticator returns the same structure as returned by the authenticatorGetAssertion method. The numberOfCredentials member is omitted.

6.3.1. Client Logic

If client receives numberOfCredentials member value exceeding 1 in response to the authenticatorGetAssertion call:

1. Call authenticatorGetNextAssertion numberOfCredentials minus 1 times.
   - Make sure 'rp' member matches the current request.
   - Remember the 'response' member.
   - Add credential user information to the 'credentialInfo' list.
2. Draw a UX that displays credentialInfo list.
3. Let user select which credential to use.
4. Return the value of the 'response' member associated with the user choice.
5. Discard all other responses.

6.4. authenticatorGetInfo (0x04)

Using this method, platforms can request that the authenticator report a list of its supported protocol versions and extensions, its AAGUID, and other aspects of its overall capabilities. Platforms should use this information to tailor their command parameters choices.

> NOTE: The values of various authenticatorGetInfo response structure members and option IDs may change over time depending upon the commands the platform sends to the authenticator.

This method takes no inputs.

On success, the authenticator returns the following **authenticatorGetInfo response structure**:

Member name	Data type	Required?	Definition
versions (0x01)	Array of strings	Required	List of supported versions. Supporte "FIDO_2_1" for CTAP2.1 / FIDO2 / Authentication authenticators, "FIDO CTAP2.0 / FIDO2 / Web Authentica "FIDO_2_1_PRE" for CTAP2.1 Prev "U2F_V2" for CTAP1/U2F authentic
extensions (0x02)	Array of strings	Optional	List of supported extensions.
aaguid (0x03)	Byte String	Required	The claimed AAGUID. 16 bytes in le the same as MakeCredential Authe specified in [WebAuthn].
options (0x04)	Map	Optional	List of supported options.

Member-name	Data type	Required?	Definition
maxMsgSize (0x05)	Unsigned Integer	Optional	Maximum message size supported authenticator.
pinUvAuthProtocols (0x06)	Array of Unsigned Integers	Optional	List of supported PIN/UV auth proto decreasing authenticator preference contain duplicate values nor be emp
maxCredentialCountInList (0x07)	Unsigned Integer	Optional	Maximum number of credentials sup credentialID list at a time by the autl be greater than zero if present.
maxCredentialIdLength (0x08)	Unsigned Integer	Optional	Maximum Credential ID Length sup authenticator. MUST be greater tha
transports (0x09)	Array of strings	Optional	List of supported transports. Values AuthenticatorTransport enum in [We MUST NOT include duplicate value present. Platforms MUST tolerate u
algorithms (0x0A)	Array of PublicKeyCredentialParameters	Optional	List of supported algorithms for crec as specified in [WebAuthn]. The arr most preferred to least preferred an include duplicate entries nor be emp PublicKeyCredentialParameters' alg are values that SHOULD be registe COSE Algorithms registry [IANA-CO
maxSerializedLargeBlobArray (0x0B)	Unsigned Integer	Optional	The maximum size, in bytes, of the blob array that this authenticator ca authenticatorLargeBlobs command MUST be specified. Otherwise it MU specified, the value MUST be ≥ 102 bytes is the least amount of storage must make available for per-creden blob arrays if it supports the large, p feature. This value is not specified a the authenticator implements the lar
forcePINChange (0x0C)	Boolean	Optional	If this member is: ↪ **present and set to `true`** getPinToken and getPinUvAuthTokenUsingP will return errors until after Change. ↪ **present and set to `false`, or a** no PIN Change is required
minPINLength (0x0D)	Unsigned Integer	Optional	This specifies the **current minimun** Unicode code points, the authentica ClientPIN. This is applicable for Clie minPINLength member MUST be al clientPin option ID is absent; it MUS authenticator supports authenticato The default **pre-configured minimu** at least 4 Unicode code points. Auth have a pre-configured default minPI than 4 code points in certain offering minPINLength reverts to its original value. Authenticators MAY also hav **configured list of RP IDs authoriz** current minimum PIN length value v minPinLength extension.
firmwareVersion (0x0E)	Unsigned Integer	Optional	Indicates the firmware version of the model identified by AAGUID. When code change to the authenticator fir authenticator MUST increase the ve
maxCredBlobLength (0x0F)	Unsigned Integer	Optional	Maximum credBlob length in bytes s authenticator. Must be present if, ar is included in the supported `extens` this value MUST be at least 32 byte
			This specifies the max number of R

Member name	Data type	Required?	Definition
maxRPIDsForSetMinPINLength (0x10)	Unsigned Integer	Optional	authenticator will accept via setMinP subcommand. The platform MUST N than this number of RP ID to the set subcommand. This is in addition to authenticator may have. If the authe support adding additional RP IDs, it MUST ONLY be present if, and only authenticator supports the setMinPI subcommand.
preferredPlatformUvAttempts (0x11)	Unsigned Integer. (CBOR major type 0)	Optional	This specifies the preferred number the getPinUvAuthTokenUsingUvWit subCommand the platform may atte back to the getPinUvAuthTokenUsingPinWithPe subCommand or displaying an error than zero. If the value is 1 then all u internal and the platform MUST only getPinUvAuthTokenUsingUvWithPe subCommand a single time. If the v authenticator MUST only decremen each iteration.
uvModality (0x12)	Unsigned Integer. (CBOR major type 0)	Optional	This specifies the user verification n by the authenticator via authenticat getPinUvAuthTokenUsingUvWithPe subcommand. This is a hint to help construct user dialogs. The values a [FIDORegistry] Section 3.1 User Ve Combining multiple bit-flags from th allowed. If clientPin is supported it M included in the bit-flags, as clientPIN user verification method.
certifications (0x13)	Map	Optional	This specifies a list of authenticator
remainingDiscoverableCredentials (0x14)	Unsigned Integer	Optional	If this member is present it indicates number of additional discoverable c be stored. If this value is zero then p create non-discoverable credentials This estimate SHOULD be based o that all future discoverable credentia maximally-sized fields and SHOULD an attempt to create a discoverable due to lack of space, even if it's pos specific request might succeed. For specific request might include fields than the maximum possible size an this value should be zero if a reques sized fields would fail. Also, a specif have an `rp.id` and `user.id` that ma discoverable credential and thus ov value should be set assuming that v
vendorPrototypeConfigCommands (0x15)	Array of Unsigned Integers	Optional	If present the authenticator supports authenticatorConfig vendorPrototyp and its value is a list of authenticato vendorCommandId values supporte empty.
attestationFormats (0x16)	Array of strings	See definition	List of supported attestation formats that support multiple attestation form `"none"`, MUST set this field. Otherw Values are taken from the "WebAut Statement Format Identifiers" regist [IANA-WebAuthn-Registries] establi [RFC8809]. The list MUST NOT incl values nor be empty if present. Plat tolerate unknown values. Support fo attestation is implied and MUST be

Member name	Data type	Required?	Definition
uvCountSinceLastPinEntry (0x17)	Unsigned Integer. (CBOR major type 0)	Optional	If present the number of internal Us operations since the last pin entry ir attempts. This allows the platform tc prompt the user for PIN on a biomet don't forget the PIN. This is optional and the interval is at the discretion c
longTouchForReset (0x18)	Boolean	Optional	If present the authenticator requires for reset.
encIdentifier (0x19)	Byte String	Optional	The value is a byte value containing is the AES-128-CBC encryption of (identifier) using HKDF-SHA-256(sal IKM = persistentPinUvAuthToken, L "encIdentifier"). The encryption iv m for each output of getInfo.
transportsForReset (0x1A)	Array of strings	Optional	List of transports that support there Values are taken from the Authentic enum in [WebAuthn]. The list MUST duplicate values nor be empty if pre MUST tolerate unknown values.
pinComplexityPolicy (0x1B)	Boolean	Optional	If present, whether the authenticato additional **current PIN complexity** minPINLength. PIN complexity polic authenticators are listed in the FIDC authenticator may have a **pre-confi complexity policy value** that is apı
pinComplexityPolicyURL (0x1C)	Byte String	Optional	If present, a URL that the platform c the user more information about the policy.
maxPINLength (0x1D)	Unsigned Integer	Optional	This specifies the **maximum PIN le** code points, the authenticator enfor An authenticator setting this value s the PIN to be represented in 63 or f applicable for ClientPIN only: the ma member MUST be absent if the clie not supported. If the authenticator s authenticatorClientPIN and the max member is absent, the effective defa is 63 code points. If specified, the maximum PIN lengt 8 Unicode code points. Authenticat pre-configured default maxPINLeng code points in certain offerings. UTF points may be represented by 1-4 o maximum length passed in the PIN always be less than 63 octets.

All options are in the form key-value pairs with string IDs and boolean values. When anoption ID is not present, the default is applied per table below. The following table lists all defined option IDs as of CTAP version "FIDO_2_2":

Option ID	Definition	Default
plat	platform device: Indicates that the device is attached to the client and therefore can't be removed and used on another client.	false
rk	Specifies whether this authenticator can create discoverable credentials, and therefore can satisfy	false

Option ID	Definition	Default
	authenticatorGetAssertion requests with the allowList parameter omitted.	
clientPin	ClientPIN feature support: If present and set to true, it indicates that the device is capable of accepting a PIN from the client and PIN has been set. If present and set to false, it indicates that the device is capable of accepting a PIN from the client and PIN has not been set yet. If absent, it indicates that the device is not capable of accepting a PIN from the client. ClientPIN is one of the overall ways to do user verification, although ClientPIN is not considered a built-in user verification method.	Not supported
up	user presence: Indicates that the device is capable of testing user presence.	true
uv	user verification: Indicates that the authenticator supports a built-in user verification method. For example, devices with UI, biometrics fall into this category. If present and set to true, it indicates that the device is capable of built-in user verification and its user verification feature is presently configured. If present and set to false, it indicates that the authenticator is capable of built-in user verification and its user verification feature is not presently configured. For example, an authenticator featuring a built-in biometric user verification feature that is not presently configured will return this "uv" option id set to false. If absent, it indicates that the authenticator does not have a built-in user verification capability. A device that can only do Client PIN will not return the "uv" option id. If a device is capable of both built-in user verification and Client PIN, the authenticator will return both the "uv" and the "clientPin" option ids.	Not Supported
pinUvAuthToken	If pinUvAuthToken is: ↪ **present and set to `true`** if the clientPin option id is present and set to `true`, then the authenticator supports authenticatorClientPIN's getPinUvAuthTokenUsingPinWithPermissions subcommand. If the uv option id is present and set to `true`, then the authenticator supports authenticatorClientPIN's getPinUvAuthTokenUsingUvWithPermissions subcommand. ↪ **present and set to `false`, or absent.** the authenticator does not support authenticatorClientPIN's getPinUvAuthTokenUsingPinWithPermissions and getPinUvAuthTokenUsingUvWithPermissions subcommands.	Not Supported
	If this noMcGaPermissionsWithClientPin is: ↪ **present and set to `true`** A pinUvAuthToken obtained via getPinUvAuthTokenUsingPinWithPermissions (or getPinToken) cannot be used for authenticatorMakeCredential or	false

Option ID	Definition	Default
noMcGaPermissionsWithClientPin	authenticatorGetAssertion commands, because it will lack the necessary mc and ga permissions. In this situation, platforms SHOULD NOT attempt to use getPinUvAuthTokenUsingPinWithPermissions if using getPinUvAuthTokenUsingUvWithPermissions fails. ↪ **present and set to `false`, or absent.** A pinUvAuthToken obtained via getPinUvAuthTokenUsingPinWithPermissions (or getPinToken) can be used for authenticatorMakeCredential or authenticatorGetAssertion commands. Note: noMcGaPermissionsWithClientPin MUST only be present if the clientPin option ID is present.	
largeBlobs	If largeBlobs is: ↪ **present and set to `true`** the authenticator supports the authenticatorLargeBlobs command. ↪ **present and set to `false`, or absent.** The authenticatorLargeBlobs command is NOT supported. This option MUST NOT be set to `true` if the largeBlob extension is supported instead.	Not supported
ep	Enterprise Attestation feature support: If ep is: ↪ **Present and set to `true`** The authenticator is enterprise attestation capable, and **enterprise attestation is enabled**. ↪ **Present and set to `false`** The authenticator is enterprise attestation capable, and **enterprise attestation is disabled**. ↪ **Absent** The Enterprise Attestation feature is NOT supported.	Not supported.
bioEnroll	If bioEnroll is: ↪ **present and set to `true`** the authenticator supports the authenticatorBioEnrollment commands, and has at least one bio enrollment presently provisioned. ↪ **present and set to `false`** the authenticator supports the authenticatorBioEnrollment commands, and does not yet have any bio enrollments provisioned. ↪ **absent** the authenticatorBioEnrollment commands are NOT supported.	Not Supported
userVerificationMgmtPreview	"FIDO_2_1_PRE" Prototype Bio enrollment support: If userVerificationMgmtPreview is: ↪ **present and set to `true`** the authenticator supports the Prototype authenticatorBioEnrollment (0x40) commands, and has at least one bio enrollment presently provisioned. ↪ **present and set to `false`** the authenticator supports the Prototype	Not Supported

Option ID	Definition	Default
	authenticatorBioEnrollment (0x40) commands, and does not yet have any bio enrollments provisioned. ↪ **absent** the Prototype authenticatorBioEnrollment (0x40) commands are not supported.	
uvBioEnroll	getPinUvAuthTokenUsingUvWithPermissions support for requesting the be permission: This option ID MUST only be present ifbioEnroll is also present. If uvBioEnroll is: ↪ **present and set to `true`** requesting the be permission when invoking getPinUvAuthTokenUsingUvWithPermissions is supported. ↪ **present and set to `false`, or absent.** requesting the be permission when invoking getPinUvAuthTokenUsingUvWithPermissions is NOT supported.	Not Supported
authnrCfg	authenticatorConfig command support: If authnrCfg is: ↪ **present and set to `true`** the authenticatorConfig command is supported. ↪ **present and set to `false`, or absent.** the authenticatorConfig command is NOT supported.	Not Supported
uvAcfg	getPinUvAuthTokenUsingUvWithPermissions support for requesting the acfg permission: This option ID MUST only be present ifauthnrCfg is also present. If uvAcfg is: ↪ **present and set to `true`** requesting the acfg permission when invoking getPinUvAuthTokenUsingUvWithPermissions is supported. ↪ **present and set to `false`, or absent.** requesting the acfg permission when invoking getPinUvAuthTokenUsingUvWithPermissions is NOT supported.	Not Supported
credMgmt	Credential management support: If credMgmt is: ↪ **present and set to `true`** the authenticatorCredentialManagement command is supported. ↪ **present and set to `false`, or absent.** the authenticatorCredentialManagement command is NOT supported.	Not Supported
	Credential management Read Only support: If perCredMgmtRO is: ↪ **present and set to `true`** requesting the pcmr permission when invoking getPinUvAuthTokenUsingUvWithPermissions or	Not Supported

Option ID	Definition	Default
perCredMgmtRO	getPinUvAuthTokenUsingPinWithPermissions is supported. ↪ **present and set to `false`, or absent.** requesting the pcmr permission when invoking getPinUvAuthTokenUsingUvWithPermissions or getPinUvAuthTokenUsingPinWithPermissions is NOT supported.	
credentialMgmtPreview	"FIDO_2_1_PRE" Prototype Credential management support: If credentialMgmtPreview is: ↪ **present and set to `true`** the Prototype authenticatorCredentialManagement (0x41) command is supported. ↪ **present and set to `false`, or absent.** the Prototype authenticatorCredentialManagement (0x41) command is NOT supported.	Not Supported
setMinPINLength	Support for the Set Minimum PIN Length feature. If setMinPINLength is: ↪ **present and set to `true`** the setMinPINLength subcommand is supported. ↪ **present and set to `false`, or absent.** the setMinPINLength subcommand is NOT supported. Note: setMinPINLength MUST only be present if the clientPin option ID is present.	Not Supported
makeCredUvNotRqd	Support for making non-discoverable credentials without requiring User Verification. If `makeCredUvNotRqd` is: ↪ **present and set to `true`** the authenticator allows creation of non-discoverable credentials without requiring any form of user verification, if the platform requests this behaviour. ↪ **present and set to `false`, or absent.** the authenticator requires some form of user verification for creating non-discoverable credentials, regardless of the parameters the platform supplies for the authenticatorMakeCredential command. Authenticators SHOULD include this option with the value `true`.	false
alwaysUv	Support for the Always Require User Verification feature: If `alwaysUv` is ↪ **present and set to `true`** the authenticator supports the Always Require User Verification feature and it is **enabled**. ↪ **present and set to `false`** the authenticator supports the Always Require User Verification feature but it is **disabled**. ↪ **absent**	Not Supported

Option ID	Definition	Default
	the authenticator does not support the Always Require User Verification feature NOTE: If the alwaysUv option ID is present and `true` the authenticator MUST set the value of makeCredUvNotRqd to `false`.	

6.5. authenticatorClientPIN (0x06)

This command exists so that plaintext PINs are not sent to the authenticator. Instead, a **PIN/UV auth protocol** (aka **pinUvAuthProtocol**) ensures that PINs are encrypted when sent to an authenticator and are exchanged for a pinUvAuthToken that serves to authenticate subsequent commands. Additionally, authenticators supporting built-in user verification methods can provide a pinUvAuthToken upon user verification.

The **pinUvAuthToken** and **persistentPinUvAuthToken** are randomly-generated, opaque bytestrings that are large enough to be effectively unguessable. See § 6.5.2.1 pinUvAuthToken State for details.

Two PIN/UV auth protocols are defined herein:

- § 6.5.6 PIN/UV Auth Protocol One
- § 6.5.7 PIN/UV Auth Protocol Two

Each PIN/UV auth protocol:

- maintains its own pinUvAuthToken and persistentPinUvAuthToken so that no unexpected, cross-protocol interactions occur, and
- is a concrete instantiation of § 6.5.4 PIN/UV Auth Protocol Abstract Definition.

NOTE: The platform MAY flexibly manage the lifetime of its copy of the pinUvAuthToken based on the usage scenario. However, it SHOULD erase its copy of the pinUvAuthToken as soon as possible when it is no longer needed. The authenticator can also expire the pinUvAuthToken based on certain conditions such as changing a PIN, authenticator timeouts, when returning CTAP2_ERR_OPERATION_DENIED or CTAP2_ERR_CREDENTIAL_EXCLUDED errors, the platform system waking up from a suspend state, the platform sending commands with no optional pinUvAuthParam, etc. If the pinUvAuthToken has expired, the authenticator will return CTAP2_ERR_PIN_AUTH_INVALID and the platform can act on the error accordingly, e.g., by getting a new pinUvAuthToken from the authenticator.

NOTE: The authenticator is only required to manage one pinUvAuthToken, though it MAY manage one per transport interface in the case that it supports multiple simultaneous transport protocols.

6.5.1. PIN Composition Requirements

Platforms MUST enforce the following, baseline, requirements on PINs used with this specification:

- Minimum PIN Length: 4 Unicode characters
- Maximum PIN Length: UTF-8 representation MUST NOT exceed 63 bytes
- PIN are in Unicode normalization form C.
- PIN MUST NOT end in a 0x00 byte

Authenticators MUST enforce the following, baseline, requirements on PINs:

- Minimum PIN Length: 4 code points.

  NOTE: Authenticators can enforce a greater minimum length.
- Maximum PIN Length: 63 bytes
- PIN storage on the device has to provide the same, or better, security assurances as provided for private keys.

Note: [FIPS140-3] references "memorized secret" requirements from SP 800-63B section 5.1.1.2. The latter states that at AAL2 and above:

> "Any memorized secret used by the authenticator for activation SHALL be a randomly-chosen numeric value at least 6 decimal digits in length or other memorized secret [at least 8 ASCII or Unicode characters in length]."

This specification attempts to count code points as an *approximation* of Unicode characters. It is understood that some scripts have multiple code points per character and may need to have additional procedural controls to conform with [FIPS140-3] or other security standards.

6.5.2. PIN/UV Auth Protocol Global State

Authenticators keep the following global state, independent of any specific PIN/UV auth protocol:

6.5.2.1. pinUvAuthToken State

A pinUvAuthToken has the following associated **state variables**. When initially generated via resetPinUvAuthToken(), the pinUvAuthToken's state variables are set to the initial values given below. The state variables values are managed via the interface given in § 6.5.3.2 pinUvAuthToken State Maintenance Functions.

> NOTE: The pinUvAuthToken-issuing operations call beginUsingPinUvAuthToken() to update the pinUvAuthToken's state variables' values prior to issuing the pinUvAuthToken to the platform. For example, they will use the latter function to set both or either the userVerified flag and/or the userPresent flag to `true`, and start the usage timer.

A pinUvAuthToken is associated with these state variables:

- A permissions RP ID, initially null.
- A ***permissions set*** whose possible values are those of pinUvAuthToken permissions. It is initially empty.
- A **usage timer**, initially not running.

  > NOTE: Once running, the timer is observed by pinUvAuthTokenUsageTimerObserver().

- An **in use flag**, initially set to `false`, meaning that the pinUvAuthToken is **not in use**. When the in use flag is set to `true`, the pinUvAuthToken is said to be **in use**.
- A **initial usage time limit**, initially not set. beginUsingPinUvAuthToken() sets this value according to the transport the platform is using to communicate with it. The platform MUST invoke an authenticator operation using the pinUvAuthToken within this time limit for the pinUvAuthToken to remain valid for the full max usage time period. The default maximum per-transport initial usage time limit values are:
  - usb: 30 seconds
  - nfc: 19.8 seconds (16 bit counter with 3311hz clock: max time before overflow)
  - ble: 30 seconds
  - internal: 30 seconds

  Authenticators MAY use other values that are less than the default maximum values.

  Authenticators MAY implement a **rolling timer**, initialized to the per-transport initial usage time limit, where the pinUvAuthToken and its state variables remain valid as long as the platform again uses the pinUvAuthToken in an operation before the rolling timer expires. If so, the rolling timer is again initialized to the initial usage time limit. This continues until the max usage time period expires. See pinUvAuthTokenUsageTimerObserver().

  > NOTE: Authenticators should utilize the rolling timer approach judiciously, e.g., because some features, such as authenticatorBioEnrollment and authenticatorCredentialManagement, may need to accommodate infrequent user interactions. Thus the rolling timer approach may be most applicable to authenticatorMakeCredential and authenticatorGetAssertion operations.

- A **user present time limit** defining the length of time the user is considered "present", as represented by the userPresent flag, after user presence is collected. The user present time limit defaults to the same default maximum per-transport values as the initial usage time limit, although authenticators MAY use other values that are less than the default maximum values, including zero.

  > NOTE: The user present time limit value of zero accommodates the case where an authenticator does not wish to support maintaining "user present" state (i.e., "cached user presence").

- A **max usage time period** value, which SHOULD default to a maximum of 10 minutes (600 seconds), though authenticators MAY use other values less than the latter default, possibly depending upon the use case, e.g., which transport is in use.
- A **userVerified flag**, initially `false`.
- A **userPresent flag**, initially `false`.

6.5.2.2. PersistentPinUvAuthToken State

When initially generated via resetPersistentPinUvAuthToken(), the persistentPinUvAuthToken's state variables are set to the initial values given below.

A persistentPinUvAuthToken is associated with these **state variables**:

- A ***permissions set*** whose possible values are those of pinUvAuthToken permissions. It is initially empty.

6.5.2.3. PIN-Entry and User Verification Retries Counters

1. **pinRetries** counter:
   - pinRetries counter is an unsigned integer, representing the number of attempts left before PIN is disabled.
   - Authenticators MUST allow no more than 8 retries but MAY set a lower maximum.
   - Each correct PIN entry resets the pinRetries and the uvRetries counters back to their maximum values unless the PIN is already disabled.
   - Each incorrect PIN entry decrements the pinRetries by 1.
   - Once the pinRetries counter reaches 0, both ClientPin as well as built-in user verification are disabled and can only be enabled if authenticator is reset.
2. **uvRetries** counter:
   - The uvRetries counter is an unsigned integer, representing the number of user verification attempts left before built-in user verification is disabled.
   - **maxUvRetries** is a global value statically configured into an authenticator; it is the maximum number of retries that a user can experience. uvRetries is initialized to this value. Its value MUST be in the range of 1 to 25, inclusive.

     > NOTE: This value is determined by the authenticator vendor based on the desired FIDO security certification level. This limit protects against brute force attacks. It is the total number of attempts allowed for all built-in user verification methods.

   - **maxUvAttemptsForInternalRetries** is a global value configured into an authenticator. It is the maximum number of times the authenticator will retry internally when internalRetry is true as part of the performBuiltInUv() algorithm. This is used for older platforms when the "uv" parameter is set as `true` OR when an authenticator vendor wants the platform to try calling it only once as indicated by the preferredPlatformUvAttempts value. If preferredPlatformUvAttempts is 1, maxUvAttemptsForInternalRetries value MUST be in range of 1 to maxUvRetries inclusive. If preferredPlatformUvAttempts is *NOT* 1, maxUvAttemptsForInternalRetries value MUST be in range of 1 to 5 inclusive.
   - Once the uvRetries counter reaches 0, built-in user verification MUST be disabled and can only be re-enabled if the authenticator is reset or the correct clientPIN is provided via the authenticatorClientPIN's getPinUvAuthTokenUsingPinWithPermissions or getPinToken subCommands.
   - **internalRetry** is a authenticator-internal boolean parameter. It defaults to `false`. It is explicitly set to `true` if the authenticator intends to perform multiple internal uv retries before returning an error to the platform.

6.5.3. Utility Functions

These utility functions are independent of the particular PIN/UV auth protocol in use.

6.5.3.1. Perform Built-in User Verification Algorithm

performBuiltInUv(internalRetry) → success | error:

1. If internalRetry is `true` then let *attemptsBeforeReturning* be set to `maxUvAttemptsForInternalRetries`.
2. Else let *attemptsBeforeReturning* be set to 1.
3. If clientPIN is `true` and pinRetries is 0, then let the uvRetries counter be set to 0 and return error.
4. If uvRetries is 0 then return error.
5. Decrement the uvRetries counter by 1.

   > NOTE: It is best practice to decrement the counter before performing built-in user verification. This prevents some hardware attacks that could provide an attacker with a unlimited number of presentation attempts. If the sample input times out the authenticator may re-increment the uvRetries counter to its previous value, if no matching is preformed by the authenticator. Some platforms will send authenticatorGetAssertion requests in parallel to multiple authenticators causing the ones not touched by the user to decrement uvRetries to 0 over time unless the uvRetries is re-incremented to the previous value after an input time out.

6. Decrement *attemptsBeforeReturning* by 1.
7. Perform built-in user verification.
8. If a user action timeout occurs, return `error`.
9. If built-in user verification succeeds then set the uvRetries counter to maxUvRetries and return `success`.
10. Else (built-in user verification failed), if *attemptsBeforeReturning* > 0, go to Step 4.
11. Otherwise, return `error`.

6.5.3.2. pinUvAuthToken State Maintenance Functions

beginUsingPinUvAuthToken(userIsPresent)

This function prepares the pinUvAuthToken for use by the platform, which has invoked one of the pinUvAuthToken-issuing operations, by setting particular pinUvAuthToken state variables to given use-case-specific values. See also § 6.5.5.7 Operations to Obtain a pinUvAuthToken.

1. Set the userPresent flag to the value of userIsPresent.
2. Set the userVerified flag to `true`.
3. Set the initial usage time limit to a transport-specific value, as described in § 6.5.2.1 pinUvAuthToken State.
4. Start the pinUvAuthToken usage timer, set the in use flag to `true`, and assign pinUvAuthTokenUsageTimerObserver() to observe the usage timer. The pinUvAuthToken is now *in use*.

pinUvAuthTokenUsageTimerObserver()

This function observes the pinUvAuthToken usage timer and takes appropriate action upon the specified conditions:

1. If the usage timer is not running, return.
2. While the overall usage timer has not reached the max usage time period, perform the following substeps:
    1. If the current user present time limit is reached, call clearUserPresentFlag().
    2. If the initial usage time limit is reached without the platform using the pinUvAuthToken in an authenticator operation then call stopUsingPinUvAuthToken(), and terminate these steps.
    3. If the authenticator does not utilize a rolling timer then continue.
    4. If the authenticator utilizes a rolling timer then:
        1. If the platform uses the pinUvAuthToken in an authenticator operation before the rolling timer expires then:
            1. Set the rolling timer to the applicable initial usage time limit and continue.
        2. Otherwise (implying the rolling timer expires) call stopUsingPinUvAuthToken(), and terminate these steps.
3. Call stopUsingPinUvAuthToken(), and terminate these steps.

getUserPresentFlagValue() → userPresentFlagValue

1. If the pinUvAuthToken is *in use* then set the userPresentFlagValue to the current value of the pinUvAuthToken's userPresent flag.
2. Otherwise (implying a pinUvAuthToken exists and is *not in use*, or does not exist), set userPresentFlagValue to `false`.

    > NOTE: The pinUvAuthToken may not exist because the pinUvAuthToken feature is not in use or is not supported.

3. Return userPresentFlagValue.

getUserVerifiedFlagValue() → userVerifiedFlagValue

1. If the pinUvAuthToken is *in use* then set the userVerifiedFlagValue to the current value of the pinUvAuthToken's userVerified flag.
2. Otherwise (implying a pinUvAuthToken exists and is *not in use*, or does not exist), set userVerifiedFlagValue to `false`.

    > NOTE: The pinUvAuthToken may not exist because the pinUvAuthToken feature is not in use or is not supported.

3. Return userVerifiedFlagValue.

clearUserPresentFlag()

1. If the pinUvAuthToken is *in use* then set the pinUvAuthToken's userPresent flag to `false`, otherwise do nothing.

clearUserVerifiedFlag()

1. If the pinUvAuthToken is *in use* then set the pinUvAuthToken's userVerified flag to `false`, otherwise do nothing.

clearPinUvAuthTokenPermissionsExceptLbw()

1. If the pinUvAuthToken is *in use* then clear all of the pinUvAuthToken's permissions, except for lbw, otherwise do nothing.

stopUsingPinUvAuthToken()

1. Set all of the pinUvAuthToken's state variables to their initial values as given in § 6.5.2.1 pinUvAuthToken State.

> Note: This causes the pinUvAuthToken's in use flag to be set to `false`, denoting the pinUvAuthToken as *not in use*.
>
> pinUvAuthToken that are *not in use* MUST NOT validate when verified in the context of the Prototype authenticatorBioEnrollment or Prototype authenticatorCredentialManagement commands.

6.5.4. PIN/UV Auth Protocol Abstract Definition

A specific PIN/UV auth protocol defines an implementation of two interfaces to cryptographic services: one for the authenticator, and one for the platform.

The authenticator interface is:

initialize()
: This process is run by the authenticator at power-on.

regenerate()
: Generates a fresh public key.

resetPinUvAuthToken()
: Generates a fresh pinUvAuthToken.

resetPersistentPinUvAuthToken()
: Generates a fresh persistentPinUvAuthToken.

getPublicKey() → coseKey
: Returns the authenticator's public key as a COSE_Key structure.

decapsulate(peerCoseKey) → sharedSecret | error
: Processes the output of encapsulate from the peer and produces a shared secret, known to both platform and authenticator.

decrypt(sharedSecret, ciphertext) → plaintext | error
: Decrypts a ciphertext, using sharedSecret as a key, and returns the plaintext.

verify(key, message, signature) → success | error
: Verifies that the signature is a valid MAC for the given message. If the key parameter value is the current pinUvAuthToken, it also checks whether thepinUvAuthToken is in use or not.

The platform interface is:

initialize()
: This is run by the platform when starting a series of transactions with a specific authenticator.

encapsulate(peerCoseKey) → (coseKey, sharedSecret) | error
: Generates an encapsulation for the authenticator's public key and returns the message to transmit and the shared secret.

encrypt(key, demPlaintext) → ciphertext
: Encrypts a plaintext to produce a ciphertext, which may be longer than the plaintext. The plaintext is restricted to being a multiple of the AES block size (16 bytes) in length.

decrypt(key, ciphertext) → plaintext | error
: Decrypts a ciphertext and returns the plaintext.

authenticate(key, message) → signature
: Computes a MAC of the given message.

(In the pseudocode function definitions, above, a function takes a number of arguments that are given in parentheses and yields a result that is one of the types separated by a bar ('|'). If a function doesn't yield any meaningful result then it implicitly yields a value of the unit type, written "success", which carries no information.)

The following PIN/UV auth protocols, specified herein, define concrete instantiations of the above interfaces:

- § 6.5.6 PIN/UV Auth Protocol One
- § 6.5.7 PIN/UV Auth Protocol Two

6.5.5. authenticatorClientPIN (0x06) Command Definition

This authenticatorClientPIN command allows a platform to use aPIN/UV auth protocol to perform a number of actions:

- Performing key agreement to obtain the shared secret
- Setting a PIN
- Changing a PIN
- Obtaining the pinUvAuthToken

The command takes the following input parameters:

Parameter name	Data type	Required?	Definition
pinUvAuthProtocol	Unsigned		PIN/UV protocol version chosen by the platform. This MUST be a value supported by the

Parameter name	Data type	Required?	Definition
(0x01)	Integer	Optional	authenticator, as determined by the pinUvAuthProtocols field of the authenticatorGetInfo response.
subCommand (0x02)	Unsigned Integer	Required	The specific action being requested.
keyAgreement (0x03)	COSE_Key	Optional	The platform key-agreement key. This COSE_Key-encoded public key MUST contain the optional "alg" parameter and MUST NOT contain any other optional parameters. The "alg" parameter MUST contain a COSEAlgorithmIdentifier value.
pinUvAuthParam (0x04)	Byte String	Optional	The output of calling authenticate on some context specific to the subcommand.
newPinEnc (0x05)	Byte String	Optional	An encrypted PIN.
pinHashEnc (0x06)	Byte String	Optional	An encrypted proof-of-knowledge of a PIN.
permissions (0x09)	Unsigned Integer	Optional	Bitfield of permissions. If present, MUST NOT be 0. See § 6.5.5.7 Operations to Obtain a pinUvAuthToken.
rpId (0x0A)	String	Optional	The RP ID to assign as the permissions RP ID.

The authenticatorClientPIN subCommands are:

subCommand Name	subCommand Number
getPINRetries	0x01
getKeyAgreement	0x02
setPIN	0x03
changePIN	0x04
getPinToken (superseded by getPinUvAuthTokenUsingUvWithPermissions or getPinUvAuthTokenUsingPinWithPermissions, thus for backwards compatibility only)	0x05
getPinUvAuthTokenUsingUvWithPermissions	0x06
getUVRetries	0x07
getPinUvAuthTokenUsingPinWithPermissions	0x09

On success, authenticator returns the following structure in its response:

Parameter name	Data type	Required?	Definition
KeyAgreement (0x01)	COSE_Key	Optional	The result of the authenticator calling getPublicKey. Used to convey the authenticator's public key to the platform so that the platform can call encapsulate. This COSE_Key-encoded public key MUST contain the optional "alg" parameter and MUST NOT contain any other optional parameters. The "alg" parameter MUST contain a COSEAlgorithmIdentifier value.
pinUvAuthToken (0x02)	Byte String	Optional	The pinUvAuthToken, encrypted by calling encrypt with the shared secret as the key.
pinRetries (0x03)	Unsigned Integer	Optional	Number of PIN attempts remaining before lockout. This is optionally used to show in UI when collecting the PIN in setting a new PIN, changing existing PIN and obtaining a pinUvAuthToken flows.
powerCycleState (0x04)	Boolean	Optional	Present and `true` if the authenticator requires a power cycle before any future PIN operation, `false` if no power cycle needed. If the field is omitted, no information is given about whether a power cycle is needed or not. This field is only valid in response to a getRetries request and authenticators MUST NOT use this field as an alternative to returning CTAP2_ERR_PIN_AUTH_BLOCKED when that is

Parameter name	Data type	Required?	Definition
			required by this specification: the power cycle behaviour is a security property and cannot be delegated to the platform to enforce.
uvRetries (0x05)	Unsigned Integer	Optional	Number of uv attempts remaining before lockout.

6.5.5.1. Authenticator Configuration Operations Upon Power Up

At power-up, the authenticator calls initialize for each pinUvAuthProtocol that it supports.

6.5.5.2. Platform getting PIN retries from Authenticator

PIN retries count is the number of PIN attempts remaining before PIN is disabled on the device. When the PIN retries count nears zero, the platform can optionally warn the user to be careful while entering the PIN.

Platform performs the following operations to get pinRetries:

1. Platform sends authenticatorClientPIN command with following parameters to the authenticator:
    1. subCommand: getPINRetries(0x01)
2. Authenticator responds back with pinRetries and, optionally, powerCycleState.

6.5.5.3. Platform getting UV Retries from Authenticator

UV retries count is the number of built-in UV attempts remaining before built-in UV is disabled on the device. When the UV retries count nears zero, the platform can optionally warn the user to be careful while performing user verification.

Platform performs the following operations to get uvRetries:

1. Platform sends authenticatorClientPIN command with following parameters to the authenticator:
    1. subCommand: getUVRetries(0x07)
2. Authenticator responds back with uvRetries.

6.5.5.4. Obtaining the Shared Secret

Platforms obtain a shared secret for each transaction. The authenticator does not have to keep a list of sharedSecrets for all active sessions. If there are subsequent authenticatorClientPIN transactions, a new sharedSecret is generated every time.

Platform performs the following operations to arrive at the sharedSecret:

1. The platform selects a mutually supported PIN/UV auth protocol by considering the list of protocols supported by the authenticator, as reported in the pinUvAuthProtocols member of the authenticatorGetInfo response. If there are multiple mutually supported protocols, and the platform has no preference, it SHOULD select the one listed first in pinUvAuthProtocols.
2. The platform sends authenticatorClientPIN command with following parameters to the authenticator:
    1. pinUvAuthProtocol: as chosen above
    2. subCommand: getKeyAgreement(0x02)
3. If the authenticator does not receive mandatory parameters for this subcommand, end the operation by returning CTAP2_ERR_MISSING_PARAMETER.
4. If the authenticator does not support the selected pinUvAuthProtocol, it returns CTAP1_ERR_INVALID_PARAMETER.
5. Otherwise the authenticator sends a response with the following parameters:
    1. keyAgreement: the result of calling getPublicKey for the selected pinUvAuthProtocol.
6. The platform calls encapsulate with the public key that the authenticator returned in order to generate the **platform key-agreement key** and the **shared secret**.

6.5.5.5. Setting a New PIN

The following operations are performed to set up a new PIN:

> The below applies to both § 6.5.5.5 Setting a New PIN and § 6.5.5.6 Changing existing PIN:
>
> An arbitrary Unicode character corresponds to one or more Unicode code points. While the platform enforces a user-visible limit of at least four Unicode characters for the PIN length (e.g., by counting grapheme clusters), this results in actually collecting at the very minimum four Unicode code points, and perhaps (many) more, depending on the script employed.

1. The platform collects the new PIN (*newPinUnicode*) from the user as Unicode characters in Normalization Form C.
2. Let *platformCollectedPinLengthInCodePoints* be the length in code points of *newPinUnicode* after normalization is applied.
    1. If the minPINLength member of the authenticatorGetInfo response is absent, then let *platformMinPINLengthInCodePoints* be 4. (The default minimum value)
    2. Else let *platformMinPINLengthInCodePoints* be the value of the minPINLength member of the authenticatorGetInfo response.
    3. If *platformCollectedPinLengthInCodePoints* is less than *platformMinPINLengthInCodePoints* then the platform SHOULD display a "PIN too short" error message to the user.
    4. Let "newPin" be the UTF-8 representation of *newPinUnicode*.
    5. If the byte length of "newPin" is greater than the max UTF-8 representation limit of 63 bytes, then the platform SHOULD display a "PIN too long" error message to the user.

    > NOTE: The platform collects the PIN before obtaining the shared secret. This prevents the shared secret from being reset if a NFC transport is used and the user removes the authenticator from the NFC reader's field while typing the PIN.
3. The Platform obtains the shared secret from the authenticator.
4. Platform sends authenticatorClientPIN command with following parameters to the authenticator:
    1. pinUvAuthProtocol: as selected when getting the shared secret.
    2. subCommand: setPIN(0x03).
    3. keyAgreement: the platform key-agreement key.
    4. newPinEnc: the result of calling encrypt(shared secret, `paddedPin`) where `paddedPin` is `newPin` padded on the right with 0x00 bytes to make it 64 bytes long. (Since the maximum length of `newPin` is 63 bytes, there is always at least one byte of padding.)
    5. pinUvAuthParam: the result of calling authenticate(shared secret, newPinEnc).
5. Authenticator performs following operations upon receiving the request:
    1. If the authenticator does not receive mandatory parameters for this command, it returns CTAP2_ERR_MISSING_PARAMETER error.
    2. If `pinUvAuthProtocol` is not supported, return CTAP1_ERR_INVALID_PARAMETER.
    3. If a PIN has already been set, authenticator returns CTAP2_ERR_PIN_AUTH_INVALID error.
    4. The authenticator calls decapsulate on the provided platform key-agreement key to obtain the shared secret. If an error results, it returns CTAP1_ERR_INVALID_PARAMETER.
    5. The authenticator calls verify(shared secret, newPinEnc, pinUvAuthParam)
        1. If an error results, it returns CTAP2_ERR_PIN_AUTH_INVALID.
    6. The authenticator calls decrypt(shared secret, newPinEnc) to produce `paddedNewPin`. If an error results, it returns CTAP2_ERR_PIN_AUTH_INVALID.
    7. If `paddedNewPin` is NOT 64 bytes long, it returns CTAP1_ERR_INVALID_PARAMETER.
    8. The authenticator drops all trailing 0x00 bytes from `paddedNewPin` to produce `newPin`.
    9. The authenticator checks the length of `newPin` against the current minimum PIN length, returning CTAP2_ERR_PIN_POLICY_VIOLATION if it is too short.
    10. An authenticator MAY impose arbitrary, additional constraints on PINs. If newPin fails to satisfy such additional constraints, the authenticator returns CTAP2_ERR_PIN_POLICY_VIOLATION.
    11. Authenticator remembers newPin length internally as **PINCodePointLength**.
    12. Authenticator stores `LEFT(SHA-256(newPin), 16)` internally as **CurrentStoredPIN**, sets the pinRetries counter to maximum count, and returns CTAP2_OK.

6.5.5.6. Changing existing PIN

The following operations are performed to change an existing PIN:

1. The Platform collects the current PIN (*curPinUnicode*) and new PIN (*newPinUnicode*) from the user as Unicode characters in Normalization Form C.

2. Let *platformCollectedNewPinLengthInCodePoints* be the length in code points of *newPinUnicode* after applying normalization.
    1. If the minPINLength member of the authenticatorGetInfo response is absent, then let *platformMinPINLengthInCodePoints* be 4. (The default minimum value)
    2. Else let *platformMinPINLengthInCodePoints* be the value of the minPINLength member of the authenticatorGetInfo response.
    3. If *platformCollectedNewPinLengthInCodePoints* is less than *platformMinPINLengthInCodePoints* then the platform SHOULD display a "PIN too short" error message to the user.
    4. Let "newPin" be the UTF-8 representation of *newPinUnicode*.
        1. If the byte length of "newPin" is greater than the max UTF-8 representation limit of 63 bytes, then the platform SHOULD display a "New PIN too long" error message to the user.
    5. Let "curPin" be the UTF-8 representation of *curPinUnicode*.
        1. If the byte length of "curPin" is greater than the max UTF-8 representation limit of 63 bytes, then the platform SHOULD display a "Current PIN too long" error message to the user.

    > NOTE: The platform collects the PIN before obtaining the shared secret. This prevents the shared secret from being reset if a NFC transport is used and the user removes the authenticator from the NFC reader's field while typing the PIN.

3. Platform obtains the shared secret from the authenticator.
4. Platform sends authenticatorClientPIN command. with following parameters to the authenticator:
    1. pinUvAuthProtocol: as selected when getting the shared secret.
    2. subCommand: changePIN(0x04).
    3. keyAgreement: the platform key-agreement key.
    4. pinHashEnc: The result of calling encrypt(shared secret, LEFT(SHA-256(curPin), 16)).
    5. newPinEnc: the result of calling encrypt(shared secret, `paddedPin`) where `paddedPin` is `newPin` padded on the right with 0x00 bytes to make it 64 bytes long. (Since the maximum length of `newPin` is 63 bytes, there is always at least one byte of padding.)
    6. pinUvAuthParam: the result of calling authenticate(shared secret, newPinEnc || pinHashEnc).
5. Authenticator performs following operations upon receiving the request:
    1. If the authenticator does not receive mandatory parameters for this command, it returns CTAP2_ERR_MISSING_PARAMETER error.
    2. If `pinUvAuthProtocol` is not supported, return CTAP1_ERR_INVALID_PARAMETER.
    3. If the pinRetries counter is 0, return CTAP2_ERR_PIN_BLOCKED error.
    4. The authenticator calls decapsulate on the provided platform key-agreement key to obtain the shared secret. If an error results, it returns CTAP1_ERR_INVALID_PARAMETER.
    5. The authenticator calls verify(shared secret, newPinEnc || pinHashEnc, pinUvAuthParam)
        1. If an error results, it returns CTAP2_ERR_PIN_AUTH_INVALID.
    6. Authenticator decrements the pinRetries counter by 1.
    7. Authenticator decrypts pinHashEnc using decrypt(shared secret, pinHashEnc) and verifies against its internal stored `LEFT(SHA-256(curPin), 16)`.
        1. If an error results, or a mismatch is detected, the authenticator performs the following operations:
            1. Calls regenerate for the selected pinUvAuthProtocol.
            2. Authenticator returns errors according to following conditions:
                1. If the pinRetries counter is 0, return CTAP2_ERR_PIN_BLOCKED error.
                2. If the authenticator sees 3 consecutive mismatches, it returns CTAP2_ERR_PIN_AUTH_BLOCKED, indicating that power cycling is needed for further operations. This is done so that malware running on the platform should not be able to block the device without user interaction.
                3. Else return CTAP2_ERR_PIN_INVALID error.
    8. Authenticator sets the pinRetries counter to maximum value.
    9. The authenticator calls decrypt(shared secret, newPinEnc) to produce `paddedNewPin`. If an error results, it returns CTAP2_ERR_PIN_AUTH_INVALID.
    10. If `paddedNewPin` is NOT 64 bytes long, it returns CTAP1_ERR_INVALID_PARAMETER.
    11. The authenticator drops all trailing 0x00 bytes from `paddedNewPin` to produce `newPin`.
    12. The authenticator checks the length of `newPin` against the current minimum PIN length, returning CTAP2_ERR_PIN_POLICY_VIOLATION if it is too short.
    13. If the forcePINChange member of the authenticatorGetInfo response is true and `LEFT(SHA-256(newPin), 16)` is equal to its internal stored `LEFT(SHA-256(curPin), 16)` then authenticator

returns CTAP2_ERR_PIN_POLICY_VIOLATION.

14. An authenticator MAY impose arbitrary, additional constraints on PINs. If newPin fails to satisfy such additional constraints, the authenticator returns CTAP2_ERR_PIN_POLICY_VIOLATION.
15. Authenticator remembers newPin length internally as PINCodePointLength.
16. Authenticator sets the value of the forcePINChange member of the authenticatorGetInfo response to `false`,
17. Authenticator stores `LEFT(SHA-256(newPin), 16)` internally as the new value of CurrentStoredPIN.
18. Authenticator sets the pinRetries counter to maximum count.
19. Authenticator calls resetPinUvAuthToken() for *all* pinUvAuthProtocols supported by this authenticator. (I.e. all existing pinUvAuthTokens are invalidated.)
20. Authenticator calls resetPersistentPinUvAuthToken() (all persistent permissions are cleared on pin change).
21. Authenticator returns CTAP2_OK.

6.5.5.7. Operations to Obtain a pinUvAuthToken

Invoking one of the below operations only has to be performed once for the lifetime of the pinUvAuthToken. Obtaining a pinUvAuthToken once allows high security without any additional roundtrips each time a subsequent authenticator operation is invoked (except for the first key-agreement phase) and its overhead is minimal.

To obtain a pinUvAuthToken, the platform SHOULD use getPinUvAuthTokenUsingUvWithPermissions, getPinUvAuthTokenUsingPinWithPermissions or getPinToken based on authenticator capabilities as returned by authenticatorGetInfo, and considering the permissions that the platform intends to request:

- getPinUvAuthTokenUsingUvWithPermissions and getPinUvAuthTokenUsingPinWithPermissions can only be used if the pinUvAuthToken Option ID is present and `true`.
- getPinUvAuthTokenUsingUvWithPermissions can only be used if the uv Option ID is present and `true`.
- getPinUvAuthTokenUsingPinWithPermissions and getPinToken can only be used if the clientPin Option ID is present and `true`.
- When requesting the be permission, getPinUvAuthTokenUsingUvWithPermissions can only be used if the uvBioEnroll Option ID is present and `true`.
- When requesting the acfg permission, getPinUvAuthTokenUsingUvWithPermissions can only be used if the uvAcfg Option ID is present and `true`.
- When requesting the mc or ga permissions, getPinUvAuthTokenUsingPinWithPermissions can only be used if the noMcGaPermissionsWithClientPin Option ID is absent or set to `false`.

When both getPinUvAuthTokenUsingUvWithPermissions and getPinUvAuthTokenUsingPinWithPermissions can be used, the platform SHOULD use getPinUvAuthTokenUsingUvWithPermissions and in case this fails, fall back to using getPinUvAuthTokenUsingPinWithPermissions.

Expected platform behavior to obtain a pinUvAuthToken is outlined in § 6.1.1 Platform Actions for authenticatorMakeCredential (non-normative) and § 6.2.1 Platform Actions for authenticatorGetAssertion (non-normative).

> NOTE: Some permissions require the presence of the `rpId` parameter, known as a **permissions RP ID**. See also § 6.5.2.1 pinUvAuthToken State.

The following **pinUvAuthToken permissions** are defined:

Permission name	Role	Value	RP ID	Definition
mc	MakeCredential	0x01	Required	This allows the pinUvAuthToken to be used for authenticatorMakeCredential operations with the provided `rpId` parameter.
ga	GetAssertion	0x02	Required	This allows the pinUvAuthToken to be used for authenticatorGetAssertion operations with the provided `rpId` parameter.
cm	Credential Management	0x04	Optional	This allows the pinUvAuthToken to be used with the authenticatorCredentialManagement command. The `rpId` parameter is optional, if it is present, the pinUvAuthToken can only be used for Credential Management operations on Credentials associated with that RP ID.
be	Bio Enrollment	0x08	Ignored	This allows the pinUvAuthToken to be used with the authenticatorBioEnrollment command. The `rpId` parameter is ignored for this

Permission name	Role	Value	RP ID	Definition
lbw	Large Blob Write	0x10	Ignored	This allows the pinUvAuthToken to be used with the authenticatorLargeBlobs command. The `rpId` parameter is ignored for this permission.
acfg	Authenticator Configuration	0x20	Ignored	This allows the pinUvAuthToken to be used with the authenticatorConfig command. The `rpId` parameter is ignored for this permission.
pcmr	Persistent Credential Management Read Only	0x40	Ignored	This allows the persistentPinUvAuthToken to be used with the authenticatorCredentialManagement command. If this permission is present, the persistentPinUvAuthToken can only be used for Read Credential Management operations on Credentials.

When a pinUvAuthToken is used with an operation that tests user presence, it is updated to remove all permissions except lbw. If lbw was not originally requested then thepinUvAuthToken becomes permission-less and cannot be used for future operations. However, the platform can fetch a fresh pinUvAuthToken in order to perform any future operations.

If authenticatorClientPIN's getPinToken subcommand is invoked, **default permissions** of mc and ga (value 0x03) are granted for the returned pinUvAuthToken. Other pinUvAuthToken permissions can only be acquired by providing the permissions parameter to the getPinUvAuthTokenUsingPinWithPermissions (0x09) or getPinUvAuthTokenUsingUvWithPermissions (0x06) subcommands.

> Note: if default permissions are used, it is possible that the permissions RP ID is not set even though it is required for some of the permissions. It will be set on first use of the pinUvAuthToken with an RP ID (for mc and ga only). default permissions are only used with the getPinToken (0x05) subcommand.

Following operations are performed to get pinUvAuthToken:

6.5.5.7.1. Getting pinUvAuthToken using getPinToken (superseded)

- Platform collects PIN from the user.

  > NOTE: The platform collects the PIN before obtaining the shared secret. This prevents the shared secret from being reset if a NFC transport is used and the user removes the authenticator from the NFC reader's field while typing the PIN.

- Platform obtains the shared secret from the authenticator.
- Platform sends authenticatorClientPIN command. with following parameters to the authenticator:
  - pinUvAuthProtocol: as selected when getting the shared secret.
  - subCommand: getPinToken (0x05).
  - keyAgreement: the platform key-agreement key.
  - pinHashEnc: the result of calling encrypt(shared secret, LEFT(SHA-256(PIN), 16)).
- Authenticator performs following operations upon receiving the request:
  - If the authenticator does not receive mandatory parameters for this command, it returns CTAP2_ERR_MISSING_PARAMETER error.
  - If `pinUvAuthProtocol` is not supported, return CTAP1_ERR_INVALID_PARAMETER.
  - If `authenticatorClientPIN`'s permissions parameter is present in the getPinToken (0x05) subcommand, return CTAP1_ERR_INVALID_PARAMETER.
  - If `authenticatorClientPIN`'s `rpId` parameter is present in the getPinToken (0x05) subcommand, return CTAP1_ERR_INVALID_PARAMETER.
  - If the pinRetries counter is 0, return CTAP2_ERR_PIN_BLOCKED error.
  - The authenticator calls decapsulate on the provided platform key-agreement key to obtain the shared secret. If an error results, it returns CTAP1_ERR_INVALID_PARAMETER.
  - If the authenticator has a display, request user consent for thedefault permissions. If this is not approved, return CTAP2_ERR_OPERATION_DENIED.
  - Authenticator decrements the pinRetries counter by 1.
  - Authenticator decrypts pinHashEnc using decrypt and verifies against its internally stored CurrentStoredPIN.
    - If an error results, or a mismatch is detected, the authenticator performs the following operations:

- Calls regenerate for the selected pinUvAuthProtocol.
- Authenticator returns errors according to following conditions:
  - If the pinRetries counter is 0, return CTAP2_ERR_PIN_BLOCKED error.
  - If the authenticator sees 3 consecutive mismatches, it returns CTAP2_ERR_PIN_AUTH_BLOCKED, indicating that power cycling is needed for further operations. This is done so that malware running on the platform should not be able to block the device without user interaction.
  - Else return CTAP2_ERR_PIN_INVALID error.

- Authenticator sets the pinRetries counter to maximum value.
- If the value of the forcePINChange member of the authenticatorGetInfo response is `true`, authenticator returns CTAP2_ERR_PIN_INVALID error.

  NOTE: The above error value is for backwards compatibility with CTAP2.0 platforms where the authenticator implements the forcePINChange feature as part of the setMinPINLength command. A pinUvAuthToken MUST NOT be returned if PINCodePointLength is less than current minimum PIN length. This is intended to force a user to change their PIN to one that conforms to the current authenticator policy. A CTAP2.1 platform will check the forcePINChange member of the authenticatorGetInfo response, and not invoke this command without forcing the user to change PIN first.

- Create a new pinUvAuthToken by calling resetPinUvAuthToken() for *all* pinUvAuthProtocols supported by this authenticator. (I.e. all existing pinUvAuthTokens are invalidated.)
- Call beginUsingPinUvAuthToken(userIsPresent: `false`).
- If the noMcGaPermissionsWithClientPin option ID is present and set to `false`, or absent, then assign the pinUvAuthToken the default permissions.

  NOTE: If noMcGaPermissionsWithClientPin option ID is true, default permissions of mc and ga are not given, but the token is still used by older CTAP 2.0 platforms for userVerificationMgmtPreview and credentialMgmtPreview commands.

- The authenticator returns the encrypted pinUvAuthToken for the specified pinUvAuthProtocol, i.e. encrypt(shared secret, pinUvAuthToken).

6.5.5.7.2. Getting pinUvAuthToken using getPinUvAuthTokenUsingPinWithPermissions (ClientPIN)

This subCommand MUST be implemented if the authenticator includes both clientPin and pinUvAuthToken Option IDs set to `true` in the authenticatorGetInfo response.

1. Platform collects PIN from the user.

   NOTE: The platform collects the PIN before obtaining the shared secret. This prevents the shared secret from being reset if a NFC transport is used and the user removes the authenticator from the NFC reader's field while typing the PIN.

2. Platform obtains the shared secret from the authenticator.
3. Platform sends authenticatorClientPIN command. with following parameters to the authenticator:
   1. pinUvAuthProtocol: as selected when getting the shared secret.
   2. subCommand: getPinUvAuthTokenUsingPinWithPermissions (0x09).
   3. keyAgreement: the platform key-agreement key.
   4. pinHashEnc: the result of calling encrypt(shared secret, LEFT(SHA-256(PIN), 16)).
   5. permissions: mandatory, the permissions associated with this pinUvAuthToken.

      NOTE: The platform SHOULD request only the permissions absolutely necessary.

   6. `rpId`: Required for some permissions, optional for others.
4. Authenticator performs following operations upon receiving the request:
   1. If the authenticator does not receive mandatory parameters for this command, it returns CTAP2_ERR_MISSING_PARAMETER error.
   2. If `pinUvAuthProtocol` is not supported, return CTAP1_ERR_INVALID_PARAMETER.
   3. If the authenticator receives a permissions parameter with value 0, return CTAP1_ERR_INVALID_PARAMETER.
   4. The below statements each relate a pinUvAuthToken permission to a given state for a authenticatorGetInfo option ID. For each pinUvAuthToken permission present in the permissions parameter, if the statement corresponding to the permission is currently true, terminate these steps and return CTAP2_ERR_UNAUTHORIZED_PERMISSION. Undefined permissions present in the permissions parameter are ignored.

- cm: credMgmt is `false` or absent.
- be: bioEnroll is absent.
- lbw: largeBlobs is `false` or absent.
- acfg: authnrCfg is `false` or absent.
- mc: noMcGaPermissionsWithClientPin is present and set to true.
- ga: noMcGaPermissionsWithClientPin is present and set to true.
- pcmr: perCredMgmtRO is `false` or absent, or any other pinUvAuthToken permission is requested.

5. If the pinRetries counter is 0, return CTAP2_ERR_PIN_BLOCKED error.
6. The authenticator calls decapsulate on the provided platform key-agreement key to obtain the shared secret. If an error results, it returns CTAP1_ERR_INVALID_PARAMETER.
7. If the authenticator has a display, request user consent for the requested permissions. If this is not approved, return CTAP2_ERR_OPERATION_DENIED.
8. Authenticator decrements the pinRetries counter by 1.
9. Authenticator decrypts pinHashEnc using decrypt and verifies against its internally stored CurrentStoredPIN.
    1. If an error results, or a mismatch is detected, the authenticator performs the following operations:
        1. Calls regenerate for the selected pinUvAuthProtocol.
        2. Authenticator returns errors according to following conditions:
            1. If the pinRetries counter is 0, return CTAP2_ERR_PIN_BLOCKED error.
            2. If the authenticator sees 3 consecutive mismatches, it returns CTAP2_ERR_PIN_AUTH_BLOCKED, indicating that power cycling is needed for further operations. This is done so that malware running on the platform should not be able to block the device without user interaction.
            3. Else return CTAP2_ERR_PIN_INVALID error.
10. Authenticator sets the pinRetries counter to maximum value.
11. If the value of the forcePINChange member of the authenticatorGetInfo response is `true`, authenticator returns CTAP2_ERR_PIN_POLICY_VIOLATION. Platform on receiving such error response SHOULD direct the user to change the PIN.
12. If the value of the requested permissions is pcmr:
    1. Assign pcmr permission to the persistentPinUvAuthToken.
    2. The authenticator returns the encrypted persistentPinUvAuthToken for the specified pinUvAuthProtocol, i.e. encrypt(shared secret, persistentPinUvAuthToken).
13. Create a new pinUvAuthToken by calling resetPinUvAuthToken() for *all* pinUvAuthProtocols supported by this authenticator. (I.e. all existing pinUvAuthTokens are invalidated.)
14. Call beginUsingPinUvAuthToken(userIsPresent: `false`).
15. Assign the requested permissions to the pinUvAuthToken, ignoring any undefined permissions.
16. If the `rpId` parameter is present, associate the permissions RP ID with the pinUvAuthToken.
17. The authenticator returns the encrypted pinUvAuthToken for the specified pinUvAuthProtocol, i.e. encrypt(shared secret, pinUvAuthToken).

6.5.5.7.3. Getting pinUvAuthToken using getPinUvAuthTokenUsingUvWithPermissions (built-in user verification methods)

This subCommand is only applicable when the authenticator supports built-in user verification methods. This subCommand MUST be implemented if the authenticator returns both uv and pinUvAuthToken option IDs set to `true` in the authenticatorGetInfo response.

1. Platform obtains the shared secret from the authenticator.
2. Platform sends authenticatorClientPIN command. with following parameters to the authenticator:
    1. pinUvAuthProtocol: as selected when getting the shared secret.
    2. subCommand: getPinUvAuthTokenUsingUvWithPermissions (0x06).
    3. keyAgreement: the platform key-agreement key.
    4. permissions: mandatory, the permissions associated with this pinUvAuthToken.

    > NOTE: The platform SHOULD request only the permissions absolutely necessary.

    5. `rpId`: Required for some permissions, optional for others.
3. Authenticator performs following operations upon receiving the request:

1. If the authenticator does not receive mandatory parameters for this command, it returns CTAP2_ERR_MISSING_PARAMETER error.
2. If `pinUvAuthProtocol` is not supported, return CTAP1_ERR_INVALID_PARAMETER.
3. If the authenticator receives a permissions parameter with value 0, return CTAP1_ERR_INVALID_PARAMETER.
4. The below statements each relate a pinUvAuthToken permission to a given state for a authenticatorGetInfo option ID. For each pinUvAuthToken permission present in the permissions parameter, if the statement corresponding to the permission is currently true, terminate these steps and return CTAP2_ERR_UNAUTHORIZED_PERMISSION. The mc and ga permissions are always considered authorized, thus they are not listed below. Undefined permissions present in the permissions are ignored.
    - cm: credMgmt is `false` or absent.
    - be: uvBioEnroll is `false` or absent.
    - lbw: largeBlobs is `false` or absent.
    - acfg: uvAcfg is `false` or absent.
    - pcmr: perCredMgmtRO is `false` or absent, or any other pinUvAuthToken permission is requested.

    > NOTE: Some authenticators with multiple built-in user verification methods may wish to support the uvBioEnroll and authnrCfg features that enable the getPinUvAuthTokenUsingUvWithPermissions subcommand to return the be and acfg permissions, allowing the platform to enroll fingerprints or perform authenticatorConfig subCommands based, e.g., on a built-in PIN or other modality.
5. If a built-in user verification method is supported but not configured, the authenticator returns CTAP2_ERR_NOT_ALLOWED.
6. If preferredPlatformUvAttempts > 1 then let internalRetry be `false`. This indicates that the platform will try invoking this sub command preferably about preferredPlatformUvAttempts times. Else let internalRetry be `true`.
7. If the uvRetries counter is 0, return CTAP2_ERR_UV_BLOCKED error.
8. If the authenticator has a display, request user consent for the requested permissions. If this is not approved, return CTAP2_ERR_OPERATION_DENIED.
9. Let uvState be the result of calling performBuiltInUv(internalRetry)
10. If uvState is `error`:
    1. If the error reason is a user action timeout, then return CTAP2_ERR_USER_ACTION_TIMEOUT.
    2. If the uvRetries counter is 0, return CTAP2_ERR_UV_BLOCKED.
    3. Otherwise, return CTAP2_ERR_UV_INVALID.

    > NOTE: The platform, upon receipt of CTAP2_ERR_UV_INVALID, SHOULD check the uvRetries value using authenticatorClientPIN's getUVRetries subCommand. If uvRetries > 0 and preferredPlatformUvAttempts > 1, platforms can materialize a UI to inform the user (if appropriate) of the number of remaining retries remaining before user verification is blocked, in conjunction with retrying getPinUvAuthTokenUsingUvWithPermissions. If either the platform receives CTAP2_ERR_UV_BLOCKED or uvRetries is 0, and clientPin option ID is set to `true`, then the platform MAY fall back to invoking getPinUvAuthTokenUsingPinWithPermissions.
11. If the value of the requested permissions is pcmr:
    1. Assign pcmr permission to the persistentPinUvAuthToken.
    2. The authenticator returns the encrypted persistentPinUvAuthToken for the specified pinUvAuthProtocol, i.e. encrypt(shared secret, persistentPinUvAuthToken).
12. Create a new pinUvAuthToken by calling resetPinUvAuthToken() for *all* pinUvAuthProtocols supported by this authenticator. (I.e. all existing pinUvAuthTokens are invalidated.)
13. If the employed built-in user verification method supplied evidence of user interaction, then call beginUsingPinUvAuthToken(userIsPresent: `true`).

    > NOTE: Whether or not a particular built-in user verification method supplies user presence can vary between authenticators.
14. Otherwise (implying that user presence was not collected), call beginUsingPinUvAuthToken(userIsPresent: `false`).
15. Assign the requested permissions to the pinUvAuthToken, ignoring any undefined permissions.
16. If the `rpId` parameter is present, use its value as the permissions RP ID and associate it with the pinUvAuthToken.
17. The authenticator returns the encrypted pinUvAuthToken for the specified pinUvAuthProtocol, i.e. encrypt(shared secret, pinUvAuthToken).

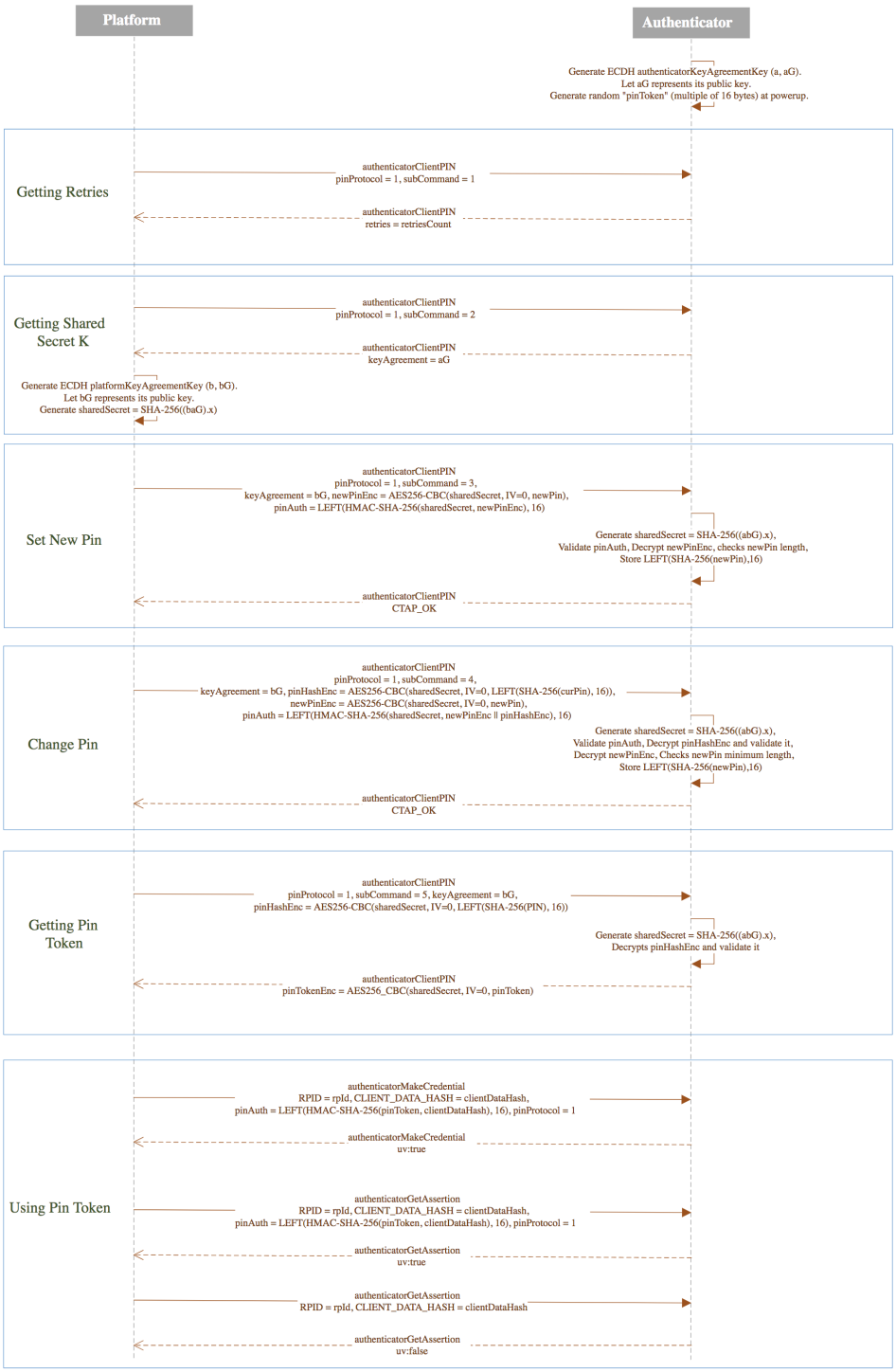

***Figure 1** Client PIN*

6.5.6. PIN/UV Auth Protocol One

This section specifies a concrete instance of the abstractPIN/UV auth protocol interfaces. It is given the numeric identifier 1, and that is the value to pass in the`pinUvAuthProtocol` parameter in various commands, to select it.

> NOTE: This PIN protocol was essentially defined in CTAP2.0, the difference between the original definition and this updated definition is that originally the `pinToken` (herein termed a pinUvAuthToken) length was unlimited. The definition given here states specific lengths for pinUvAuthTokens in both this PIN/UV Auth Protocol 1, and in PIN/UV Auth Protocol 2.

This PIN/UV auth protocol maintains the following state:

- **Key agreement key**: a P-256 private key, *x*, and the associated **public point** *x***B**, which is the result of a scalar-multiplication of the P-256 base point, **B**, by the private key.
- pinUvAuthToken, a random, opaque byte string that MUST be either 16 or 32 bytes long. This is generated afresh at power-on and reset when specified below.

This PIN/UV auth protocol defines the following internal functions:

ecdh(peerCoseKey) → sharedSecret | error

1. Parse peerCoseKey as specified for getPublicKey, below, and produce a P-256 point, **Y**. If unsuccessful, or if the resulting point is not on the curve, return error.
2. Calculate *x***Y**, the *shared point*. (I.e. the scalar-multiplication of the peer's point, **Y**, with the local private key agreement key.)
3. Let Z be the 32-byte, big-endian encoding of the x-coordinate of the shared point.
4. Return kdf(Z).

kdf(Z) → sharedSecret
Return SHA-256(Z)

(See [RFC6090] Section 4.1 and appendix (C.2) of [SP800-56A] for more ECDH key agreement protocol details and key representation.)

The operations of PIN/UV auth protocol 1 are defined as follows:

initialize()
Calls regenerate followed by resetPinUvAuthToken.

regenerate()
Generate a fresh, random P-256 private key, *x*, and compute the associated public point.

resetPinUvAuthToken()

1. Generate a fresh, random, pinUvAuthToken of either 16 or 32 bytes in length.
2. Associate pinUvAuthToken state variables with the new pinUvAuthToken, initialized per § 6.5.2.1 pinUvAuthToken State.

getPublicKey()
Return a COSE_Key with the following header parameters:

- 1 (kty) = 2 (EC2)
- 3 (alg) = -25 (although this is not the algorithm actually used)
- -1 (crv) = 1 (P-256)
- -2 (x) = 32-byte, big-endian encoding of the x-coordinate of *x***B** (the key agreement key's public point)
- -3 (y) = 32-byte, big-endian encoding of the y-coordinate of *x***B**

encapsulate(peerCoseKey) → (coseKey, sharedSecret) | error

1. Let sharedSecret be the result of calling ecdh(peerCoseKey). Return any resulting error.
2. Return (getPublicKey(), sharedSecret)

decapsulate(peerCoseKey) → sharedSecret | error
Return ecdh(peerCoseKey)

encrypt(key, demPlaintext) → ciphertext
Return the AES-256-CBC encryption of demPlaintext using an all-zero IV. (No padding is performed as the size of demPlaintext is required to be a multiple of the AES block length.)

decrypt(key, demCiphertext) → plaintext | error
If the size of demCiphertext is not a multiple of the AES block length, return error. Otherwise return the AES-256-CBC decryption of demCiphertext using an all-zero IV.

authenticate(key, message) → signature
Return the first 16 bytes of the result of computing HMAC-SHA-256 with the given key and message.

verify(key, message, signature) → success | error

1. If the key parameter value is the current pinUvAuthToken and it is not in use, then return error.
2. Compute HMAC-SHA-256 with the given key and message. Return success if signature is 16 bytes and is equal to the first 16 bytes of the result, otherwise return error.

6.5.7. PIN/UV Auth Protocol Two

This section provides a PIN/UV auth protocol that is intended to aid FIPS [CMVP] certification of authenticators. It is given the numeric identifier `2`, and that is the value to pass in the `pinUvAuthProtocol` parameter in various commands, to select it.

> NOTE: support for this is mandatory in some cases. See § 9 Mandatory features.

The length of the pinUvAuthToken for PIN/UV auth protocol two MUST be 32 bytes. Otherwise, it inherits all the behavior of PIN protocol one and overrides only these functions:

kdf(Z) → sharedSecret
Return
HKDF-SHA-256(salt = 32 zero bytes, IKM = Z, L = 32, info = "CTAP2 HMAC key") ||
HKDF-SHA-256(salt = 32 zero bytes, IKM = Z, L = 32, info = "CTAP2 AES key")
(see [RFC5869] for the definition of HKDF).

> NOTE: This is two separate invocations of HKDF whose results are concatenated together. It can NOT be equivalently performed using a single invocation with L=64.

resetPinUvAuthToken()

1. Generate a fresh, random, 32-byte, pinUvAuthToken.
2. Associate pinUvAuthToken state variables with the new pinUvAuthToken, initialized per § 6.5.2.1 pinUvAuthToken State.

encrypt(key, demPlaintext) → ciphertext

1. Discard the first 32 bytes of key. (This selects the AES-key portion of the shared secret.)
2. Let iv be a 16-byte, random bytestring.
3. Let ct be the AES-256-CBC encryption of demPlaintext using key and iv. (No padding is performed as the size of demPlaintext is required to be a multiple of the AES block length.)
4. Return iv || ct.

decrypt(key, demCiphertext) → plaintext | error

1. Discard the first 32 bytes of key. (This selects the AES-key portion of the shared secret.)
2. If demPlaintext is less than 16 bytes in length, return an error
3. Split demPlaintext after the 16th byte to produce two subspans, iv and ct.
4. Return the AES-256-CBC decryption of ct using key and iv.

authenticate(key, message) → signature

1. If key is longer than 32 bytes, discard the excess. (This selects the HMAC-key portion of the shared secret. When key is the pinUvAuthToken, it is exactly 32 bytes long and thus this step has no effect.)
2. Return the result of computing HMAC-SHA-256 on key and message.

verify(key, message, signature) → success | error

1. If the key parameter value is the current pinUvAuthToken and it is not in use, then return error.
2. If key is longer than 32 bytes, discard the excess. (This selects the HMAC-key portion of the shared secret. When key is the pinUvAuthToken, it is exactly 32 bytes long and thus this step has no effect.)
3. Compute HMAC-SHA-256 with the given key and message. Return success if the signature is equal to the result, otherwise return an error.

6.5.8. PRF values used

Throughout this protocol, the pseudo-random function defined by HMAC-SHA-256 and the pinUvAuthToken is evaluated for various values in order to authenticate requests from the platform. It is important that these values uniquely identify the salient parameters of the requests that they authenticate otherwise a PRF output from one context could be observed by an attacker and replayed in a different context.

(It is a known weakness that, within the scope of a single pinUvAuthToken value, requests may be reordered or replayed by an attacker.)

For clarity, all the patterns of values used by this protocol are enumerated in the following table:

Context	Pattern of PRF argument
authenticatorMakeCredential	32 arbitrary bytes
authenticatorGetAssertion	32 arbitrary bytes
authenticatorClientPIN	32×0xff || `0608` || 32-bit value || CBOR array
authenticatorBioEnrollment	`0101` || CBOR map `0102` || CBOR map `0104` `0105` || CBOR map

authenticatorCredentialManagement	`01`
	`02`
	`04` || CBOR map
	`06` || CBOR map

authenticatorLargeBlobs	32×0xff || `0c00` || 32-bit value || SHA-256(contents of `set` byte string, i.e. *not* including an outer CBOR tag with major type two)
authenticatorConfig	32×0xff || `0d` || 8-bit value || CBOR map

In order to avoid collisions with values already used the following pattern will be used for future commands: 32 0xff bytes, followed by the command code as a single byte, followed by an unambiguous substructure defined by each command.

The leading 0xff bytes in the pattern separate the value from any possible value used in an authenticatorMakeCredential or authenticatorGetAssertion command. As motivation, consider the authenticatorBioEnrollment command which does not use this pattern. The argument to authenticatorGetAssertion is a clientDataHash which, in a WebAuthn context, is the hash of a potentially predictable JSON string containing an attacker-controlled nonce. Offline, an attacker can iterate over many nonces until they find one which will produce a clientDataHash that starts with `0101a1`, is followed by a CBOR string or integer not equal to three, and then by a CBOR value that exactly fills the remaining space. This requires around 2^{32} offline hash evaluations but, if the attacker can observe the PRF output sent by the platform for an authenticatorGetAssertion command using that nonce, then they can replay it to start a fingerprint enrollment as the PRF argument also matches the pattern for enrolling a fingerprint. (Although note that more work is required to complete the enrollment as that requires further commands to be authenticated.)

6.6. authenticatorReset (0x07)

Resetting an authenticator is a potentially destructive operation. Authenticators MAY thus choose, for each transport they support, whether this command will be supported when received on that transport. For example, an authenticator may choose not to support this command over NFC, fearing that coincidentally nearby readers may send malicious reset commands.

However this command MUST be supported on at least onetransport. If the USB HID transport is supported then this command MUST be supported on that transport.

This method is used by the client to reset an authenticator back to a factory default state. Specifically this action at least:

- Invalidates all generated credentials, including those created over CTAP1/U2F.
- Erases all discoverable credentials.
- Resets the serialized large-blob array storage, if any, to theinitial serialized large-blob array value.
- Generate a new 128-bit random value for the device identifier.
- Disables those features that are denoted as being subject to disablement by authenticatorReset:
  - Enterprise attestation
- Resets those features that are denoted as being subject to reset by authenticatorReset:
  - Always Require User Verification
  - Set Minimum PIN Length
  - Persistent PUAT state

Additionally:

- In order to prevent an accidental triggering of this mechanism,evidence of user interaction is required.
- In case of authenticators with no display, request MUST have come to the authenticator within 10 seconds of powering up of the authenticator.

If all conditions are met, authenticator returns CTAP2_OK. If this command is disabled for the transport used, the authenticator returns CTAP2_ERR_OPERATION_DENIED. If user presence is explicitly denied, the authenticator returns CTAP2_ERR_OPERATION_DENIED. If a user action timeout occurs, the authenticator returns CTAP2_ERR_USER_ACTION_TIMEOUT. If the request comes after 10 seconds of powering up, the authenticator returns CTAP2_ERR_NOT_ALLOWED.

6.7. authenticatorBioEnrollment (0x09)

This command is used by the platform to provision/enumerate/delete bio enrollments in the authenticator.

It takes the following input parameters:

Parameter name	Data type	Required?	Definition
modality (0x01)	Unsigned Integer	Optional	The user verification modality being requested
subCommand (0x02)	Unsigned Integer	Optional	The authenticator user verification sub command currently being requested
subCommandParams (0x03)	CBOR Map	Optional	Map of subCommands parameters. This parameter MAY be omitted when the subCommand does not take any arguments.
pinUvAuthProtocol (0x04)	Unsigned Integer	Optional	PIN/UV protocol version chosen by the platform.
pinUvAuthParam (0x05)	Byte String	Optional	The output of calling authenticate on some context specific to the subcommand.
getModality (0x06)	Boolean	Optional	Get the user verification type modality. This MUST be set to true.

The type of modalities supported are as under:

modality Name	modality Number
fingerprint	0x01

The list of sub commands for fingerprint(0x01) modality is:

subCommand Name	subCommand Number
enrollBegin	0x01
enrollCaptureNextSample	0x02
cancelCurrentEnrollment	0x03
enumerateEnrollments	0x04
setFriendlyName	0x05
removeEnrollment	0x06
getFingerprintSensorInfo	0x07

subCommandParams Fields:

Field name	Data type	Required?	Definition
templateId (0x01)	Byte String	Optional	Template Identifier.
templateFriendlyName (0x02)	String	Optional	Template Friendly Name.
timeoutMilliseconds (0x03)	Unsigned Integer	Optional	Timeout in milliSeconds.

On success, authenticator returns the following structure in its response:

Parameter name	Data type	Required?	Definition
modality (0x01)	Unsigned Integer	Optional	The user verification modality.
fingerprintKind (0x02)	Unsigned Integer	Optional	Indicates the type of fingerprint sensor. For touch type sensor, its value is 1. For swipe type sensor its value is 2.
maxCaptureSamplesRequiredForEnroll (0x03)	Unsigned Integer	Optional	Indicates the maximum good samples required for enrollment.
templateId (0x04)	Byte String	Optional	Template Identifier.
lastEnrollSampleStatus (0x05)	Unsigned Integer	Optional	Last enrollment sample status.
remainingSamples (0x06)	Unsigned Integer	Optional	Number of more sample required for enrollment to complete
templateInfos (0x07)	CBOR ARRAY	Optional	Array of templateInfo's
maxTemplateFriendlyName (0x08)	Unsigned Integer	Optional	Indicates the maximum number of bytes the authenticator will accept as a templateFriendlyName.

TemplateInfo definition:

Field name	Data type	Required?	Definition
templateId (0x01)	Byte String	Required	Template Identifier.
templateFriendlyName (0x02)	String	Optional	Template Friendly Name.

lastEnrollSampleStatus types:

lastEnrollSampleStatus Name	lastEnrollSampleStatus Value	Definition
CTAP2_ENROLL_FEEDBACK_FP_GOOD	0x00	Good fingerprint capture.
CTAP2_ENROLL_FEEDBACK_FP_TOO_HIGH	0x01	Fingerprint was too high.
		Fingerprint

lastEnrollSampleStatus Name	lastEnrollSampleStatus Value	Definition
CTAP2_ENROLL_FEEDBACK_FP_TOO_LOW	0x02	was too low.
CTAP2_ENROLL_FEEDBACK_FP_TOO_LEFT	0x03	Fingerprint was too left.
CTAP2_ENROLL_FEEDBACK_FP_TOO_RIGHT	0x04	Fingerprint was too right.
CTAP2_ENROLL_FEEDBACK_FP_TOO_FAST	0x05	Fingerprint was too fast.
CTAP2_ENROLL_FEEDBACK_FP_TOO_SLOW	0x06	Fingerprint was too slow.
CTAP2_ENROLL_FEEDBACK_FP_POOR_QUALITY	0x07	Fingerprint was of poor quality.
CTAP2_ENROLL_FEEDBACK_FP_TOO_SKEWED	0x08	Fingerprint was too skewed.
CTAP2_ENROLL_FEEDBACK_FP_TOO_SHORT	0x09	Fingerprint was too short.
CTAP2_ENROLL_FEEDBACK_FP_MERGE_FAILURE	0x0A	Merge failure of the capture.
CTAP2_ENROLL_FEEDBACK_FP_EXISTS	0x0B	Fingerprint already exists.
(unused)	0x0C	(this error number is available)
CTAP2_ENROLL_FEEDBACK_NO_USER_ACTIVITY	0x0D	User did not touch/swipe the authenticator.
CTAP2_ENROLL_FEEDBACK_NO_USER_PRESENCE_TRANSITION	0x0E	User did not lift the finger off the sensor.

NOTE: In order to support the authenticator performingauthenticatorMakeCredential or authenticatorGetAssertion immediately after bio enrollment, authenticators SHOULD NOT expire the pinUvAuthToken at the completion of bio enrollment.

6.7.1. Feature detection

The bioEnroll option ID in the authenticatorGetInfo response defines feature support detection for this feature.

6.7.2. Get bio modality

Following operations are performed to get bio modality supported by the authenticator:

- Platform sends authenticatorBioEnrollment command with following parameters:
  - getModality (0x06): true.
- Authenticator returns authenticatorBioEnrollment response with following parameters:
  - modality (0x01): It represents the type of modality authenticator supports. For fingerprint, its value is 1.

6.7.3. Get fingerprint sensor info

Following operations are performed to get fingerprint sensor information:

- Platform sends authenticatorBioEnrollment command with following parameters:
  - modality (0x01): fingerprint (0x01).
  - subCommand (0x02): getFingerprintSensorInfo (0x07)

- Authenticator returns authenticatorBioEnrollment response with following parameters:
  - fingerprintKind (0x02):
    - For touch type fingerprints, its value is 1.
    - For swipe type fingerprints, its value is 2.
  - maxCaptureSamplesRequiredForEnroll (0x03): Indicates the maximum good samples required for enrollment.
  - maxTemplateFriendlyName (0x08): Indicates the maximum number of bytes the authenticator will accept as a templateFriendlyName.

6.7.4. Enrolling fingerprints

Following operations are performed to enroll a fingerprint:

- Platform gets pinUvAuthToken from the authenticator with thebe permission.
- Platform sends authenticatorBioEnrollment command with following parameters to begin the enrollment:
  - modality (0x01): fingerprint (0x01).
  - subCommand (0x02): enrollBegin (0x01).
  - subCommandParams (0x03): Map containing following parameters
    - timeoutMilliseconds (0x03) (optional): timeout in milliseconds
  - pinUvAuthProtocol (0x04): as selected when getting the shared secret.
  - pinUvAuthParam (0x05): authenticate(pinUvAuthToken, fingerprint (0x01) || enrollBegin (0x01) || subCommandParams).
- Authenticator on receiving such request performs following procedures.
  - If `pinUvAuthParam` is missing from the input map, end the operation by returning CTAP2_ERR_PUAT_REQUIRED.
  - If the authenticator does not receive mandatory parameters for this subcommand, end the operation by returning CTAP2_ERR_MISSING_PARAMETER.
  - If `pinUvAuthProtocol` is not supported, return CTAP1_ERR_INVALID_PARAMETER.
  - Authenticator calls verify(pinUvAuthToken, fingerprint (0x01) || enrollBegin (0x01) || subCommandParams, pinUvAuthParam)
    - If the verification fails, return CTAP2_ERR_PIN_AUTH_INVALID.
  - Authenticator verifies that the token has be permission, if not, it returns CTAP2_ERR_PIN_AUTH_INVALID.
  - If there is no space available, authenticator returns CTAP2_ERR_FP_DATABASE_FULL.
  - Authenticator cancels any unfinished ongoing enrollment.
  - Authenticator generates templateId for new enrollment.
  - Authenticator sends the command to the sensor to capture the sample.
  - Authenticator returns authenticatorBioEnrollment response with following parameters:
    - templateId (0x04): template identifier of the new template being enrolled.
    - lastEnrollSampleStatus (0x05) : Status of enrollment of last sample.
    - remainingSamples (0x06) : Number of sample remaining to complete the enrollment.
- Platform sends authenticatorBioEnrollment command with following parameters to continue enrollment in a loop till remainingSamples is zero or authenticator errors out with unrecoverable error or platform wants to cancel current enrollment:
  - Platform sends authenticatorBioEnrollment command with following parameters
    - modality (0x01): fingerprint (0x01).
    - subCommand (0x02): enrollCaptureNextSample (0x02).
    - subCommandParams (0x03): Map containing following parameters
      - templateId (0x01) : template identifier platform received from enrollBegin subCommand.
      - timeoutMilliseconds (0x03) (optional): timeout in milliseconds
    - pinUvAuthProtocol (0x04): as selected when getting the shared secret.
    - pinUvAuthParam (0x05): authenticate(pinUvAuthToken, fingerprint (0x01) || enrollCaptureNextSample (0x02) || subCommandParams).
  - Authenticator on receiving such request performs following procedures.
    - If `pinUvAuthParam` is missing from the input map, end the operation by returning CTAP2_ERR_PUAT_REQUIRED.

- If the authenticator does not receive mandatory parameters for this subcommand, end the operation by returning CTAP2_ERR_MISSING_PARAMETER.
- If `pinUvAuthProtocol` is not supported, return CTAP1_ERR_INVALID_PARAMETER.
- Authenticator calls verify(pinUvAuthToken, fingerprint (0x01) || enrollCaptureNextSample (0x02) || subCommandParams, pinUvAuthParam)
  - If the verification fails, return CTAP2_ERR_PIN_AUTH_INVALID.
- Authenticator verifies that the pinUvAuthToken has be permission, if not, it returns CTAP2_ERR_PIN_AUTH_INVALID.
- If there is no space available, authenticator returns CTAP2_ERR_FP_DATABASE_FULL.
- If fingerprint is already present on the sensor, authenticator waits for user to lift finger from the sensor.
- Authenticator sends the command to the sensor to capture the sample.
- Authenticator returns authenticatorBioEnrollment response with following parameters:
  - lastEnrollSampleStatus (0x05) : Status of enrollment of last sample.
  - remainingSamples (0x06) : Number of sample remaining to complete the enrollment.

6.7.5. Cancel current enrollment

Following operations are performed to cancel current enrollment:

- Platform sends authenticatorBioEnrollment command with following parameters:
  - modality (0x01): fingerprint (0x01).
  - subCommand (0x02): cancelCurrentEnrollment (0x03).
- Authenticator on receiving such command, cancels current ongoing enrollment, if any, and returns CTAP2_OK.

6.7.6. Enumerate enrollments

Following operations are performed to enumerate enrollments:

- Platform gets pinUvAuthToken from the authenticator with thebe permission.
- Platform sends authenticatorBioEnrollment command with following parameters:
  - modality (0x01): fingerprint (0x01).
  - subCommand (0x02): enumerateEnrollments (0x04).
  - pinUvAuthProtocol (0x04): as selected when getting the shared secret.
  - pinUvAuthParam (0x05): authenticate(pinUvAuthToken, fingerprint (0x01) || enumerateEnrollments (0x04)).
- Authenticator on receiving such request performs following procedures.
  - If `pinUvAuthParam` is missing from the input map, end the operation by returning CTAP2_ERR_PUAT_REQUIRED.
  - If the authenticator does not receive mandatory parameters for this subcommand, end the operation by returning CTAP2_ERR_MISSING_PARAMETER.
  - If `pinUvAuthProtocol` is not supported, return CTAP1_ERR_INVALID_PARAMETER.
  - Authenticator calls verify(pinUvAuthToken, fingerprint (0x01) || enumerateEnrollments (0x04), pinUvAuthParam)
    - If the verification fails, return CTAP2_ERR_PIN_AUTH_INVALID.
  - Authenticator verifies that the token has be permission, if not, it returns CTAP2_ERR_PIN_AUTH_INVALID.
  - If there are no enrollments existing on authenticator, it returns CTAP2_ERR_INVALID_OPTION.
  - Authenticator returns authenticatorBioEnrollment response following parameters:
    - templateInfos (0x07) : Array of templateInfo's for all the enrollments available on the authenticator.

6.7.7. Rename/Set FriendlyName

Following operations are performed to rename a fingerprint:

- Platform gets pinUvAuthToken from the authenticator with thebe permission.
- Platform sends authenticatorBioEnrollment command with following parameters:
  - modality (0x01): fingerprint (0x01).

- subCommand (0x02): setFriendlyName (0x05).
- subCommandParams (0x03): Map containing following parameters
  - templateId (0x01) : template identifier.
  - templateFriendlyName (0x02): Friendly name of the template. (The maximum size SHOULD be the lessor of 64 bytes or the value of maxTemplateFriendlyName)
- pinUvAuthProtocol (0x04): as selected when getting the shared secret.
- pinUvAuthParam (0x05): authenticate(pinUvAuthToken, fingerprint (0x01) || setFriendlyName (0x05) || subCommandParams).

- Authenticator on receiving such request performs following procedures.
  - If `pinUvAuthParam` is missing from the input map, end the operation by returning CTAP2_ERR_PUAT_REQUIRED.
  - If the authenticator does not receive mandatory parameters for this subcommand, end the operation by returning CTAP2_ERR_MISSING_PARAMETER.
  - If `pinUvAuthProtocol` is not supported, return CTAP1_ERR_INVALID_PARAMETER.
  - If templateFriendlyName is longer than specified by maxTemplateFriendlyName, return an error e.g., CTAP1_ERR_INVALID_LENGTH.
  - Authenticator calls verify(pinUvAuthToken, fingerprint (0x01) || setFriendlyName (0x05) || subCommandParams, pinUvAuthParam)
    - If the verification fails, return CTAP2_ERR_PIN_AUTH_INVALID.
  - Authenticator verifies that the token has be permission, if not, it returns CTAP2_ERR_PIN_AUTH_INVALID.
  - If there are no enrollments existing on authenticator for the passed templateId, it returns CTAP2_ERR_INVALID_OPTION.
  - If there is an existing enrollment with that identifier, rename its friendly name and return CTAP2_OK.

6.7.8. Remove enrollment

Following operations are performed to remove a fingerprint:

- Platform gets pinUvAuthToken from the authenticator with thebe permission.
- Platform sends authenticatorBioEnrollment command with following parameters:
  - modality (0x01): fingerprint (0x01).
  - subCommand (0x02): removeEnrollment (0x06).
  - subCommandParams (0x03): Map containing following parameters
    - templateId (0x01) : template identifier.
  - pinUvAuthProtocol (0x04): as selected when getting the shared secret.
  - pinUvAuthParam (0x05): authenticate(pinUvAuthToken, fingerprint (0x01) || removeEnrollment (0x06) || subCommandParams).
- Authenticator on receiving such request performs following procedures.
  - If `pinUvAuthParam` is missing from the input map, end the operation by returning CTAP2_ERR_PUAT_REQUIRED.
  - If the authenticator does not receive mandatory parameters for this subcommand, end the operation by returning CTAP2_ERR_MISSING_PARAMETER.
  - If `pinUvAuthProtocol` is not supported, return CTAP1_ERR_INVALID_PARAMETER.
  - Authenticator calls verify(pinUvAuthToken, fingerprint (0x01) || removeEnrollment (0x06) || subCommandParams, pinUvAuthParam)
    - If the verification fails, return CTAP2_ERR_PIN_AUTH_INVALID.
  - Authenticator verifies that the token has be permission, if not, it returns CTAP2_ERR_PIN_AUTH_INVALID.
  - If there are no enrollments existing on authenticator for passed templateId, it returns CTAP2_ERR_INVALID_OPTION.
  - If there is an exiting enrollment with passed in templateInfo, delete that enrollment and return CTAP2_OK.

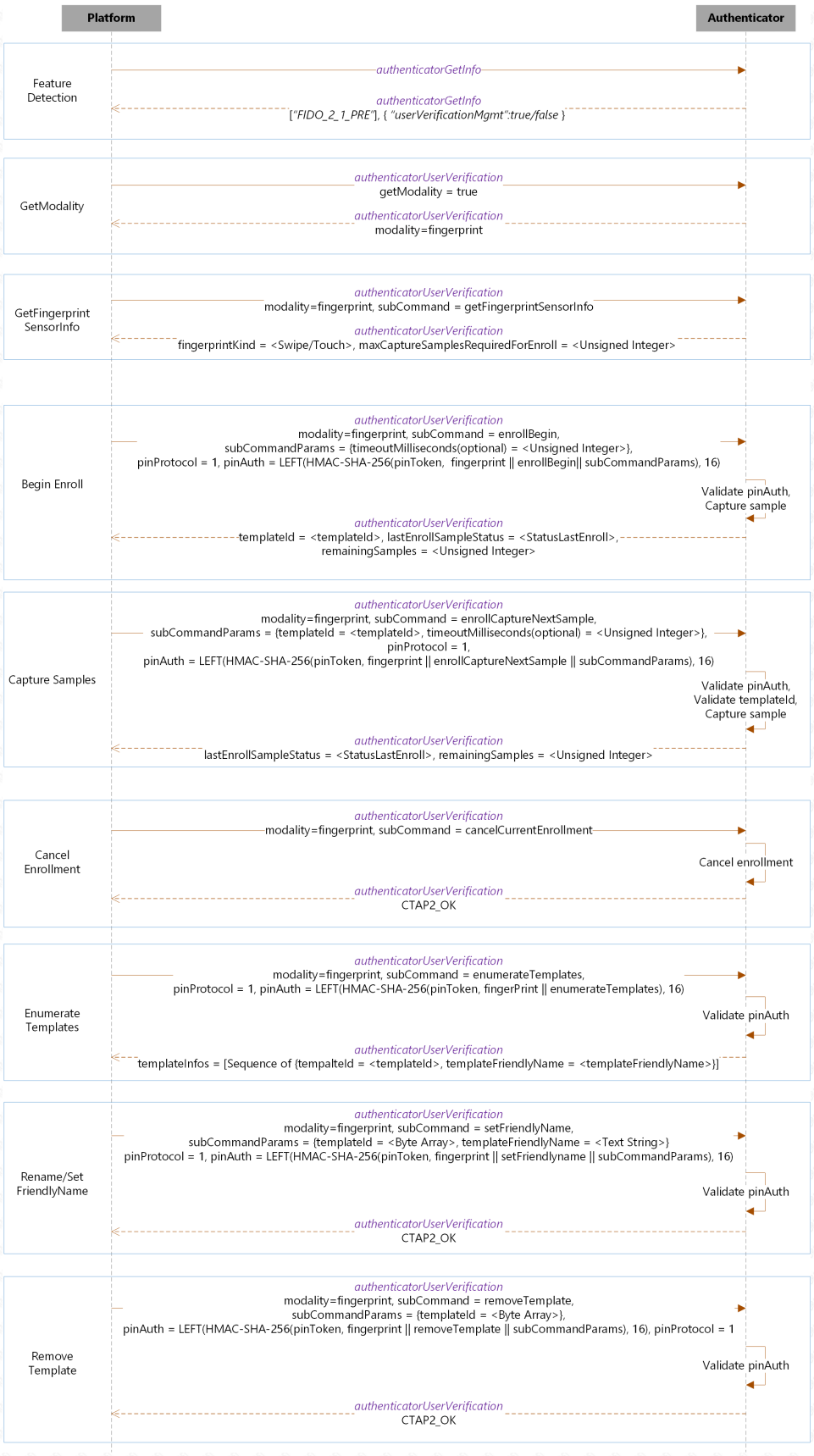

Figure 2 *User Verification Modality - Fingerprint*

6.8. authenticatorCredentialManagement (0x0A)§

This command is used by the platform to manage discoverable credentials on the authenticator.

NOTE: support for this command is mandatory in some cases. See§ 9 Mandatory features.

It takes the following input parameters:

Parameter name	Data type	Definition

Parameter name	Data type	Definition
subCommand (0x01)	Unsigned Integer	subCommand currently being requested
subCommandParams (0x02)	CBOR Map	Map of subCommands parameters.
pinUvAuthProtocol (0x03)	Unsigned Integer	PIN/UV protocol version chosen by the platform.
pinUvAuthParam (0x04)	Byte String	The output of calling authenticate on some context specific to the subcommand.

The list of sub commands for credential management is:

subCommand Name	subCommand Number
getCredsMetadata	0x01
enumerateRPsBegin	0x02
enumerateRPsGetNextRP	0x03
enumerateCredentialsBegin	0x04
enumerateCredentialsGetNextCredential	0x05
deleteCredential	0x06
updateUserInformation	0x07

subCommandParams Fields:

Field name	Data type	Definition
rpIDHash (0x01)	Byte String	RP ID SHA-256 hash
credentialID (0x02)	PublicKeyCredentialDescriptor	Credential Identifier
user (0x03)	`PublicKeyCredentialUserEntity`	User Entity

On success, authenticator returns the following structure in its response:

Parameter name	Data type	Definition
existingResidentCredentialsCount (0x01)	Unsigned Integer	Number of existing discoverable credentials present on the authenticator.
maxPossibleRemainingResidentCredentialsCount (0x02)	Unsigned Integer	Number of maximum possible remaining discoverable credentials which can be created on the authenticator.
rp (0x03)	PublicKeyCredentialRpEntity	RP Information
rpIDHash (0x04)	Byte String	RP ID SHA-256 hash
totalRPs (0x05)	Unsigned Integer	total number of RPs present on the authenticator
user (0x06)	PublicKeyCredentialUserEntity	User Information
credentialID (0x07)	PublicKeyCredentialDescriptor	PublicKeyCredentialDescriptor
publicKey (0x08)	COSE_Key	Public key of the credential.
totalCredentials (0x09)	Unsigned Integer	Total number of credentials present on the authenticator for the RP in question
credProtect (0x0A)	Unsigned Integer	Credential protection policy.
largeBlobKey (0x0B)	Byte string	Large blob encryption key.
thirdPartyPayment (0x0C)	Boolean	Whether the credential is third-party payment enabled, if supported by the authenticator

Here are some example scenarios where credential management might be used:

- The platform may want to do actual credential management, e.g. list, update, or delete credentials. In this case, a permissions RP ID is not associated with the pinUvAuthToken and all credentials can be enumerated and retrieved.
- The platform may need to fetch the public key of a credential for use in some protocols like SSH. When making the authenticatorGetAssertion request, a permissions RP ID is present (because it is required for the ga permission) but now the cm permission will only allow you to retrieve credentials related to that authenticatorGetAssertion request. This works because you do not need access to all credentials, just the ones relevant for the request's associated RP ID.
- The platform may want to garbage collect large-blobs because it finds that there is insufficient space to store a desired blob. Since it's possible that a credential has been deleted without also deleting its large blob, the platform may be able to free up enough space with garbage collection. In this case, additional user interaction may be needed because a permissions RP ID needs to be associated with the pinUvAuthToken for the ga or mc permission to be obtained, but a full enumeration needs the cm permission without any RP ID limitation. Thus the user may need to perform user verification a second time if garbage collection of just the single RP ID is insufficient.

6.8.1. Feature detection

The credMgmt option ID in the authenticatorGetInfo response defines feature support detection for this feature.

6.8.2. Getting Credentials Metadata

Following operations are performed to get credentials metadata information :

- Platform gets pinUvAuthToken from the authenticator with the cm or pcmr permission, and MUST NOT include a permissions RP ID parameter.
- Platform sends authenticatorCredentialManagement command with following parameters:
  - subCommand (0x01): getCredsMetadata (0x01).
  - pinUvAuthProtocol (0x03): as selected when getting the shared secret.
  - pinUvAuthParam (0x04): authenticate(pinUvAuthToken, getCredsMetadata (0x01)).
- Authenticator on receiving such request performs following procedures.
  - If `pinUvAuthParam` is missing from the input map, end the operation by returning CTAP2_ERR_PUAT_REQUIRED.
  - If the authenticator does not receive mandatory parameters for this subcommand, end the operation by returning CTAP2_ERR_MISSING_PARAMETER.
  - If `pinUvAuthProtocol` is not supported, return CTAP1_ERR_INVALID_PARAMETER.
  - Authenticator calls verify(persistentPinUvAuthToken, getCredsMetadata (0x01), pinUvAuthParam).
  - If pinUvAuthParam verification succeeds. (platform used persistentPinUvAuthToken)
    1. The authenticator verifies that the persistentPinUvAuthToken has the pcmr permission. If not, return CTAP2_ERR_PIN_AUTH_INVALID.
  - Else: (try to validate with pinUvAuthToken)
    1. Authenticator calls verify(pinUvAuthToken, getCredsMetadata (0x01), pinUvAuthParam).
    2. If pinUvAuthParam verification fails, authenticator returns CTAP2_ERR_PIN_AUTH_INVALID error.
    3. The authenticator verifies that the pinUvAuthToken has the cm permission and no associated permissions RP ID. If not, return CTAP2_ERR_PIN_AUTH_INVALID.
  - Authenticator returns authenticatorCredentialManagement response with following parameters:
    - existingResidentCredentialsCount (0x01) : total number of discoverable credentials existing on the authenticator.
    - maxPossibleRemainingResidentCredentialsCount (0x02) : maximum number of possible remaining discoverable credentials that can be created on the authenticator. Note that this number is an estimate as actual space consumed to create a credential depends on various conditions such as which algorithm is picked, user entity information etc.

6.8.3. Enumerating RPs

Following operations are performed to enumerate RPs present on the authenticator:

- Platform gets pinUvAuthToken from the authenticator with the cm or pcmr permission, and MUST NOT include a permissions RP ID parameter.
- Platform sends authenticatorCredentialManagement command with following parameters:

- subCommand (0x01): enumerateRPsBegin (0x02).
- pinUvAuthProtocol (0x03): as selected when getting the shared secret.
- pinUvAuthParam (0x04): authenticate(pinUvAuthToken, enumerateRPsBegin (0x02)).

- Authenticator on receiving such request performs following procedures.
  - If `pinUvAuthParam` is missing from the input map, end the operation by returning CTAP2_ERR_PUAT_REQUIRED.
  - If the authenticator does not receive mandatory parameters for this subcommand, end the operation by returning CTAP2_ERR_MISSING_PARAMETER.
  - If `pinUvAuthProtocol` is not supported, return CTAP1_ERR_INVALID_PARAMETER.
  - Authenticator calls verify(persistentPinUvAuthToken, enumerateRPsBegin (0x02), pinUvAuthParam).
  - If pinUvAuthParam verification succeeds. (platform used persistentPinUvAuthToken)
    1. The authenticator verifies that the persistentPinUvAuthToken has the pcmr permission. If not, return CTAP2_ERR_PIN_AUTH_INVALID.
  - Else: (try to validate with PinUvAuthToken)
    1. Authenticator calls verify(pinUvAuthToken, enumerateRPsBegin (0x02), pinUvAuthParam).
    2. If pinUvAuthParam verification fails, authenticator returns CTAP2_ERR_PIN_AUTH_INVALID error.
    3. The authenticator verifies that the pinUvAuthToken has the cm permission and no associated permissions RP ID. If not, return CTAP2_ERR_PIN_AUTH_INVALID.
  - If no discoverable credentials exist on this authenticator, return CTAP2_ERR_NO_CREDENTIALS.
  - Authenticator returns an authenticatorCredentialManagement response with following parameters:
    - rp (0x03): `PublicKeyCredentialRpEntity`, where the `id` field SHOULD be included and other fields MAY be included. (See § 6.8.7 Truncation of relying party identifiers about possible truncation of the `id` field and [WebAuthn] about other fields.)
    - rpIDHash (0x04) : RP ID SHA-256 hash.
    - totalRPs (0x05) : Total number of RPs present on the authenticator.
- Platform on receiving more than 1 totalRPs, performs following procedure for (totalRPs - 1) number of times:
  - Platform sends authenticatorCredentialManagement command with following parameters:
    - subCommand (0x01): enumerateRPsGetNextRP (0x03).

> NOTE: this is a stateful command and the specified implementation accommodations apply to it.

  - Authenticator on receiving such enumerateCredentialsGetNext subCommand returns authenticatorCredentialManagement response with following parameters:
    - rp (0x03): `PublicKeyCredentialRpEntity`
    - rpIDHash (0x04) : RP ID SHA-256 hash.

6.8.4. Enumerating Credentials for an RP

Following operations are performed to enumerate credentials for an RP:

- Platform gets pinUvAuthToken from the authenticator with thecm or pmcr permission.
- Platform sends authenticatorCredentialManagement command with following parameters:
  - subCommand (0x01): enumerateCredentialsBegin (0x04).
  - subCommandParams (0x02): Map containing following parameters
    - rpIDHash (0x01): RP ID SHA-256 hash.
  - pinUvAuthProtocol (0x03): as selected when getting the shared secret.
  - pinUvAuthParam (0x04): authenticate(pinUvAuthToken, enumerateCredentialsBegin (0x04) || subCommandParams).
- Authenticator on receiving such request performs following procedures.
  - If `pinUvAuthParam` is missing from the input map, end the operation by returning CTAP2_ERR_PUAT_REQUIRED.
  - If the authenticator does not receive mandatory parameters for this subcommand, end the operation by returning CTAP2_ERR_MISSING_PARAMETER.
  - If `pinUvAuthProtocol` is not supported, return CTAP1_ERR_INVALID_PARAMETER.
  - Authenticator calls verify(persistentPinUvAuthToken, enumerateCredentialsBegin (0x04) || subCommandParams, pinUvAuthParam).
  - If pinUvAuthParam verification succeeds. (platform used persistentPinUvAuthToken)

1. The authenticator verifies that the persistentPinUvAuthToken has the pcmr permission. If not, return CTAP2_ERR_PIN_AUTH_INVALID.

- Else: (try to validate with PinUvAuthToken)

1. Authenticator calls verify(pinUvAuthToken, enumerateCredentialsBegin (0x04) || subCommandParams, pinUvAuthParam).
2. If pinUvAuthParam verification fails, the authenticator returns a CTAP2_ERR_PIN_AUTH_INVALID error.
3. The authenticator verifies that the pinUvAuthToken has the cm permission and no associated permissions RP ID. If not, return CTAP2_ERR_PIN_AUTH_INVALID.

- If no discoverable credentials for this RP ID hash exist on this authenticator, return CTAP2_ERR_NO_CREDENTIALS.
- Authenticator returns authenticatorCredentialManagement response with following parameters:
  - user (0x06): PublicKeyCredentialUserEntity
  - credentialID (0x07): PublicKeyCredentialDescriptor
  - publicKey (0x08): public key of the credential in COSE_Key format
  - totalCredentials (0x09): total number of credentials for this RP
  - credProtect (0x0A): credential protection policy
  - largeBlobKey (0x0B): the contents, if any, of the storedlargeBlobKey.
  - thirdPartyPayment (0x0C): present only if the authenticator supports thethirdPartyPayment extension. True if the credential isthird-party payment enabled, false otherwise.

- Platform on receiving more than 1 totalCredentials, performs following procedure for (totalCredentials - 1) number of times:
  - Platform sends authenticatorCredentialManagement command with following parameters:
    - subCommand (0x01): enumerateCredentialsGetNextCredential (0x05).

> NOTE: this is a stateful command and the specified implementation accommodations apply to it.

  - Authenticator on receiving such enumerateCredentialsGetNext subCommand returns with following parameters:
    - user (0x06): PublicKeyCredentialUserEntity
    - credentialID (0x07): PublicKeyCredentialDescriptor
    - publicKey (0x08): public key of the credential in COSE_Key format
    - credProtect (0x0A): credential protection policy
    - largeBlobKey (0x0B): the contents, if any, of the storedlargeBlobKey.
    - thirdPartyPayment (0x0C): present only if the authenticator supports thethirdPartyPayment extension. True if the credential isthird-party payment enabled, false otherwise.

> NOTE: when enumerating credentials, platforms SHOULD take the opportunity to performlarge-blob garbage collection, if applicable.

6.8.5. DeleteCredential

Following operations are performed to delete a credential:

- Platform gets pinUvAuthToken from the authenticator with thecm permission.
- Platform sends authenticatorCredentialManagement command with following parameters:
  - subCommand (0x01): deleteCredential (0x06).
  - subCommandParams (0x02): Map containing following parameters
    - credentialId (0x02): PublicKeyCredentialDescriptor of the credential to be deleted.
  - pinUvAuthProtocol (0x03): as selected when getting the shared secret.
  - pinUvAuthParam (0x04): authenticate(pinUvAuthToken, deleteCredential (0x06) || subCommandParams).
- Authenticator on receiving such request performs following procedures.
  - If `pinUvAuthParam` is missing from the input map, end the operation by returning CTAP2_ERR_PUAT_REQUIRED.
  - If the authenticator does not receive mandatory parameters for this subcommand, end the operation by returning CTAP2_ERR_MISSING_PARAMETER.
  - If `pinUvAuthProtocol` is not supported, return CTAP1_ERR_INVALID_PARAMETER.

- Authenticator calls verify(pinUvAuthToken, deleteCredential (0x06) || subCommandParams, pinUvAuthParam)
- If pinUvAuthParam verification fails, authenticator returns CTAP2_ERR_PIN_AUTH_INVALID error.
- The authenticator verifies that the pinUvAuthToken has the cm permission and that the pinUvAuthToken does not have a permissions RP ID associated or that the pinUvAuthTokenpermissions RP ID matches the RP ID of the credential. If not, return CTAP2_ERR_PIN_AUTH_INVALID.
- If there are not credential existing matching credentialDescriptor, return CTAP2_ERR_NO_CREDENTIALS.
- Delete the credential and return CTAP2_OK.

NOTE: when deleting a credential, platforms SHOULD alsodelete any associated large blobs.

6.8.6. Updating user information

Following operations are performed to update user information associated to a credential:

- Platform gets pinUvAuthToken from the authenticator with thecm permission.
- Platform sends authenticatorCredentialManagement command with following parameters:
  - subCommand (0x01): updateUserInformation (0x07).
  - subCommandParams (0x02): Map containing the parameters that need to be updated.
    - credentialId (0x02): PublicKeyCredentialDescriptor of the credential to be updated.
    - user (0x03): a `PublicKeyCredentialUserEntity` with the updated information.
  - pinUvAuthProtocol (0x03): as selected when getting the shared secret.
  - pinUvAuthParam (0x04): authenticate(pinUvAuthToken, updateUserInformation (0x07) || subCommandParams).
- Authenticator on receiving such request performs following procedures.
  - If `pinUvAuthParam` is missing from the input map, end the operation by returning CTAP2_ERR_PUAT_REQUIRED.
  - If the authenticator does not receive mandatory parameters for this subcommand, end the operation by returning CTAP2_ERR_MISSING_PARAMETER.
  - If `pinUvAuthProtocol` is not supported, return CTAP1_ERR_INVALID_PARAMETER.
  - Authenticator calls verify(pinUvAuthToken, updateUserInformation (0x07) || subCommandParams, pinUvAuthParam)
  - If pinUvAuthParam verification fails, authenticator returns CTAP2_ERR_PIN_AUTH_INVALID error.
  - The authenticator verifies that the pinUvAuthToken has the cm permission and that the pinUvAuthToken does not have a permissions RP ID associated or that the pinUvAuthTokenpermissions RP ID matches the RP ID of the credential. If not, return CTAP2_ERR_PIN_AUTH_INVALID.
  - The authenticator searches for an existing credential matching`credentialId`.
  - If no matching credential is found, return CTAP2_ERR_NO_CREDENTIALS.
  - If the authenticator does not have enough internal storage to update the matching credential, return CTAP2_ERR_KEY_STORE_FULL.
  - If the supplied `user` parameter's `id` field is not the same as the matching credential's`id` field then return CTAP1_ERR_INVALID_PARAMETER.
  - Replace the matching credential's `PublicKeyCredentialUserEntity`'s `name`, `displayName` with the passed-in `user` details. If a field is not present in the passed`user` details, or it is present and empty, remove it from the matching credential's `PublicKeyCredentialUserEntity`.
  - Return CTAP2_OK.

6.8.7. Truncation of relying party identifiers

An authenticator MAY store relying party identifiers in order to implementauthenticatorCredentialManagement. As there is no bound on their length, authenticators MAY truncate them using a procedure that produces the same results as the code included below. If authenticators store relying party identifiers at all, they MUST store at least 32 bytes. Truncation of relying party identifiers only applies to returning a `PublicKeyCredentialRpEntity` structure in the context of this command. I.e. authenticators MUST NOT use truncated relying party identifiers for comparisons at any time, including in the context of this command.

```
#define MAX_STORED_RPID_LENGTH 32  /* MUST be >= 32 */

void maybe_truncate_rpid(uint8_t stored_rpid[MAX_STORED_RPID_LENGTH],
                         size_t *stored_len, const uint8_t *rpid,
                         size_t rpid_len) {
  if (rpid_len <= MAX_STORED_RPID_LENGTH) {
    memcpy(stored_rpid, rpid, rpid_len);
    *stored_len = rpid_len;
    return;
  }

  size_t used = 0;
  const uint8_t *colon_position = memchr(rpid, ':', rpid_len);
  if (colon_position != NULL) {
    const size_t protocol_len = colon_position - rpid + 1;
    const size_t to_copy = protocol_len <= MAX_STORED_RPID_LENGTH
                               ? protocol_len
                               : MAX_STORED_RPID_LENGTH;
    memcpy(stored_rpid, rpid, to_copy);
    used += to_copy;
  }

  if (MAX_STORED_RPID_LENGTH - used < 3) {
    *stored_len = used;
    return;
  }

  // U+2026, horizontal ellipsis.
  stored_rpid[used++] = 0xe2;
  stored_rpid[used++] = 0x80;
  stored_rpid[used++] = 0xa6;

  const size_t to_copy = MAX_STORED_RPID_LENGTH - used;
  memcpy(&stored_rpid[used], rpid + rpid_len - to_copy, to_copy);
  assert(used + to_copy == MAX_STORED_RPID_LENGTH);
  *stored_len = MAX_STORED_RPID_LENGTH;
}
```

For illutrative purposes, here are some examples of the truncation in effect:

Input RP ID	Stored RP ID	Comment
example.com	example.com	No truncation applied
myfidousingwebsite.hostingprovider.net	…ngwebsite.hostingprovider.net	Truncation applied on the left
mygreatsite.hostingprovider.info	mygreatsite.hostingprovider.info	No truncation applied to strings of length 32; any sentinel values (e.g. NUL bytes in C) are internal to the authenticator implementation and do not count towards the protocol defined length
otherprotocol://myfidousingwebsite.hostingprovider.net	otherprotocol:…ingprovider.net	Protocol strings are preserved if possible
veryexcessivelylargeprotocolname://example.com	veryexcessivelylargeprotocolname	Protocol strings may consume the entire space

6.9. authenticatorSelection (0x0B)

This command allows the platform to let a user select a certain authenticator by asking for user presence.

The command has no input parameters.

When the authenticatorSelection command is received, the authenticator will ask for user presence:

- If User Presence is received, the authenticator will return CTAP2_OK.
- If User Presence is explicitly denied by the user, the authenticator will return CTAP2_ERR_OPERATION_DENIED. The platform SHOULD NOT repeat the command for this authenticator.
- If a user action timeout occurs, the authenticator will return CTAP2_ERR_USER_ACTION_TIMEOUT. The platform MAY repeat the command for this authenticator.

If an authenticator is selected, the platform SHOULD send a cancel to all other authenticators.

6.10. authenticatorLargeBlobs (0x0C)

The credBlob extension allows for a small amount of additional, secret information to be stored with a credential. There are two options for storing a larger amount of data: this command allows a platform to store information on an authenticator and to protect credential-specific parts of it with a key that is then stored and accessed using the largeBlobKey extension. Alternatively the largeBlob extension carries the data directly in authenticatorGetAssertion requests. This section is about the former.

This command allows at least 1024 bytes of large blob data to be stored on CTAP2 authenticators. For the purposes of this command, this data is serialized as a CBOR-encoded array (called the **large-blob array**) of large-blob maps, concatenated with 16 following bytes. Those final 16 bytes are the truncated SHA-256 hash of the preceding bytes. This concatenation is referred to as the **serialized large-blob array**. The **opaque large-blob data** that is stored for a credential with this command is a byte string with RP-specific structure. This is only applicable to discoverable credentials so that garbage collection is possible.

The **initial serialized large-blob array** is the value of the serialized large-blob array on a fresh authenticator, as well as immediately after a reset. It is the byte string `h'8076be8b528d0075f7aae98d6fa57a6d3c'`, which is an empty CBOR array (`80`) followed by `LEFT(SHA-256(h'80'), 16)`.

> NOTE: the minimum length of a serialized large-blob array is 17 bytes. Omitting 16 bytes for the trailing SHA-256 hash, this leaves just one byte. This is the size of an empty CBOR array.

6.10.1. Feature detection

The largeBlobs option ID in the authenticatorGetInfo response defines feature support detection for this feature.

6.10.2. Reading and writing serialised data

The command takes the following input parameters:

Parameter name	Data type	Required?	Notes
get (0x01)	Unsigned integer	Optional	The number of bytes requested to read. MUST NOT be present if `set` is present.
set (0x02)	Byte String	Optional	A fragment to write. MUST NOT be present if `get` is present.
offset (0x03)	Unsigned integer	Required	The byte offset at which to read/write.
length (0x04)	Unsigned integer	Optional	The total length of a write operation. Present if, and only if, `set` is present and `offset` is zero.
pinUvAuthParam (0x05)	Byte String	Optional	authenticate(pinUvAuthToken, 32×0xff || h'0c00' || uint32LittleEndian(`offset`) || SHA-256(contents of `set` byte string, i.e. *not* including an outer CBOR tag with major type two))
pinUvAuthProtocol (0x06)	Unsigned integer	Optional	PIN/UV protocol version chosen by the platform.

A per-authenticator constant, `maxFragmentLength`, is here defined as the value of `maxMsgSize` (from the authenticatorGetInfo response) minus 64. The value 64 is a comfortable over-estimate of the encoding overhead of the messages defined in this section such that a byte string of length `maxFragmentLength` can be transferred without exceeding the maximum message size of the authenticator. If no `maxMsgSize` is given in the authenticatorGetInfo response) then it defaults to 1024, leaving `maxFragmentLength` to default to 960.

In addition to persistently storing the serialized large-blob array, authenticators implementing this command are required to maintain two unsigned integers in volatile memory named `expectedNextOffset` and `expectedLength`, both initially zero. This makes this command a stateful command and the specified implementation accommodations apply to it.

An authenticator performs the following actions upon receipt of this command:

1. If `offset` is not present in the input map, return CTAP1_ERR_INVALID_PARAMETER.
2. If neither `get` nor `set` are present in the input map, return CTAP1_ERR_INVALID_PARAMETER.
3. If both `get` and `set` are present in the input map, return CTAP1_ERR_INVALID_PARAMETER.
4. If `get` is present in the input map:

1. If `length` is present, return CTAP1_ERR_INVALID_PARAMETER.
2. If either of `pinUvAuthParam` or `pinUvAuthProtocol` are present, return CTAP1_ERR_INVALID_PARAMETER.
3. If the value of `get` is greater than `maxFragmentLength`, return CTAP1_ERR_INVALID_LENGTH.
4. If the value of `offset` is greater than the length of the stored serialized large-blob array, return CTAP1_ERR_INVALID_PARAMETER.
5. Return a CBOR map, as defined below, where the value of `config` is a substring of the stored serialized large-blob array. The substring SHOULD start at the offset given in `offset` and contain the number of bytes specified as `get`'s value. If too few bytes exist at that offset, return the maximum number available. Note that if `offset` is equal to the length of the serialized large-blob array then this will result in a zero-length substring.

5. Else (implying that `set` is present in the input map):
    1. If the length of the value of `set` is greater than `maxFragmentLength`, return CTAP1_ERR_INVALID_LENGTH. (The "value of `set`" means the contents of the byte string corresponding to the key `set` (0x02), *not* including the outer CBOR tag with major type two.)
    2. If the value of `offset` is zero:
        1. If `length` is not present, return CTAP1_ERR_INVALID_PARAMETER.
        2. If the value of `length` is greater than 1024 bytes and exceeds the capacity of the device, return CTAP2_ERR_LARGE_BLOB_STORAGE_FULL. (Authenticators MUST be capable of storing at least 1024 bytes.)
        3. If the value of `length` is less than 17, return CTAP1_ERR_INVALID_PARAMETER. (See note above about minimum lengths.)
        4. Set `expectedLength` to the value of `length`.
        5. Set `expectedNextOffset` to zero.
    3. Else (i.e. the value of `offset` is not zero):
        1. If `length` is present, return CTAP1_ERR_INVALID_PARAMETER.
    4. If the value of `offset` is not equal to `expectedNextOffset`, return CTAP1_ERR_INVALID_SEQ.
    5. If the authenticator is protected by some form of user verification or the alwaysUv option ID is present and `true`:
        1. If `pinUvAuthParam` is absent from the input map, then end the operation by returning CTAP2_ERR_PUAT_REQUIRED.
        2. If `pinUvAuthProtocol` is absent from the input map, then end the operation by returning CTAP2_ERR_MISSING_PARAMETER.
        3. If `pinUvAuthProtocol` is not supported, return CTAP1_ERR_INVALID_PARAMETER.
        4. The authenticator calls `verify(pinUvAuthToken, 32×0xff || h'0c00' || uint32LittleEndian(offset) || SHA-256(contents of set byte string, i.e. *not* including an outer CBOR tag with major type two), pinUvAuthParam)`.
            1. If the verification fails, return CTAP2_ERR_PIN_AUTH_INVALID.
        5. Check if the pinUvAuthToken has the lbw permission, if not, return CTAP2_ERR_PIN_AUTH_INVALID.
    6. If the sum of `offset` and the length of the value of `set` is greater than the value of `expectedLength`, return CTAP1_ERR_INVALID_PARAMETER.
    7. If the value of `offset` is zero, prepare a buffer to receive a new serialized large-blob array.
    8. Append the value of `set` to the buffer containing the pending serialized large-blob array.
    9. Update `expectedNextOffset` to be the new length of the pending serialized large-blob array.
    10. If the length of the pending serialized large-blob array is equal to `expectedLength`:
        1. Verify that the final 16 bytes in the buffer are the truncated SHA-256 hash of the preceding bytes. If the hash does not match, return CTAP2_ERR_INTEGRITY_FAILURE.
        2. Commit the contents of the buffer as the new serialized large-blob array for this authenticator.
        3. Return CTAP2_OK and an empty response.
    11. Else:
        1. More data is needed to complete the pending serialized large-blob array.
        2. Return CTAP2_OK and an empty response. Await further writes.

> NOTE: user verification is only checked above if user verification is configured on a device or the authenticator always requires some form of user verification feature is enabled. This implies that a serialized large-blob array can be written without user verification if user verification is not configured.

NOTE: To read (i.e., "get") per-credential large-blob data given a credential ID, the platform must first use an authenticatorGetAssertion operation to obtain the associated largeBlobKey in order to be able to decrypt the large-blob data (if any). Thus the confidentiality of any large-blob data associated with the credential is dependent upon the credential's protection policy. This means that even though a platform may obtain the large-blob array at will, it will be unable to obtain large-blob plaintexts if it cannot successfully perform authenticatorGetAssertion operations using the associated credential(s), e.g., without obtaining user verification. Also, the "trial decryption" approach employed for obtaining plaintext means that large-blobs do not disclose a priori the existence of credentials having a credProtect level 3 userVerificationRequired policy.

The response to a `get` request, referenced above, takes the following form:

Parameter name	Data type	Required?	Notes
config (0x01)	Byte String	Required	Contains the requested substring of the serialized large-blob array.

In order to read a serialized large-blob array, a platform is expected to first issue a request where `offset` is zero and `get` equals the value of `maxFragmentLength`, which is `maxMsgSize` – 64 bytes, as defined above. If the length of the response is equal to the value of `get` then more data may be available and the platform SHOULD repeatedly issue requests, each time updating `offset` to equal the amount of data received so far. It stops once a short (or empty) fragment is returned. Once complete, the platform MUST confirm that the embedded SHA-256 hash is correct, based on the definition above. If not, the configuration is corrupt and the platform MUST discard it and act as if the initial serialized large-blob array was received.

In order to write a serialized large-blob array, a platform is expected to first issue a request where `offset` is zero, `length` is the full length of the data to be written, and `set` contains a prefix of the data to be written, truncated at `maxFragmentLength` bytes, if `length` is greater than `maxFragmentLength`. If truncation is needed then one or more further requests are needed to complete the transfer, with `offset` updated each time to contain the amount of data written so far and `set` containing consecutive substrings of the data. The authenticator will implicitly know when the transfer is complete because of the length given in the first request.

The algorithm to be performed by the authenticator given above assumes that the authenticator double-buffers the serialized large-blob array. (I.e. it writes proposed updates into a separate buffer and only overwrites the effective config once validation has completed.) A compliant authenticator MAY be implemented using only a single buffer as follows: when appending to the buffer, use `expectedLength` to buffer the final 16 bytes of the serialized large-blob array in volatile storage. Once the transfer is complete, perform validation and only write the final 16 bytes to persistent storage if successful. This prevents the SHA-256 checksum of an invalid serialized large-blob array from being persisted.

NOTE: even with double-buffering, the copy from the temporary buffer might be interrupted, resulting in a "torn write". This will be detected by the platform when reading because the checksum won't match, but results in an unusable config. Thus double-buffering minimises the chance of corruption, but does not always eliminate it.

Despite best efforts, torn writes, platform errors, and storage corruption may result in a situation where an authenticator finds itself having stored an invalid serialized large-blob array. (I.e. the SHA-256 hash does not match.) In this case, the authenticator MAY reset the stored value with the initial serialized large-blob array.

An authenticator MUST NOT act on the contents of the serialized large-blob array except for checking the trailing hash: it is purely for platforms to adjust their behavior in response to.

Authenticators MUST set the serialized large-blob array to the initial serialized large-blob array byte string when reset.

Platforms MUST ensure that the large-blob array (i.e. without the trailing 16 bytes) is a CBOR array where all entries conform to the large-blob map structure defined below. The maps and array MUST be encoded using the canonical rules. Platforms MUST NOT attempt to write a serialized large-blob array that exceeds the `maxSerializedLargeBlobArray` reported by the authenticator in the authenticatorGetInfo response. Platforms SHOULD take care to preserve existing entries in a large-blob array where space permits. For example, platforms should read, and then insert values into, an existing large-blob array as opposed to blindly writing a fresh array.

6.10.3. Large, per-credential blobs

The elements of the large-blob array MUST conform to the following **large-blob map** structure. **Conformance**, in this context, means that a map MUST include all required elements, MAY include optional elements, and MAY include unknown elements. The values of all documented elements present MUST match the specified type and MUST comply with any additional restrictions documented for them.

Element name	Data type	Required?	Notes
ciphertext	Byte	Required	AEAD_AES_256_GCM ciphertext, implicitly including the

Element name	Data type	Required?	Notes
(0x01)	String		AEAD “authentication tag” at the end.
nonce (0x02)	Byte String	Required	AEAD_AES_256_GCM nonce. MUST be exactly 12 bytes long.
origSize (0x03)	Unsigned Integer	Required	Contains the length, in bytes, of the uncompressed data.

The `ciphertext` member contains the output of encrypting the opaque large-blob data with the AEAD_AES_256_GCM algorithm from [RFC5116]. The inputs to the AEAD are:

- Nonce: the 12-byte value from `nonce`.
- Plaintext: the compressed opaque large-blob data.
- Associated data: The value 0x626c6f62 ("blob") || uint64LittleEndian(`origSize`).
- Key: the 32-byte value stored using the largeBlobKey extension.

6.10.4. Reading per-credential large-blob data

The platform SHOULD perform the following steps in order to read the opaque large-blob data for a given credential. The platform must know the credential ID of the intended credential a priori, which it might have been given, or might have learnt from performing an authenticatorGetAssertion operation without an allowList parameter.

1. If the authenticator does not support the largeBlobKey extension, as defined in that section, return an error.
2. Perform an authenticatorGetAssertion operation with "largeBlobKey": `true` in the `extensions` map in order to fetch the largeBlobKey for the credential. (This step may be skipped if the pertinent output is already known.)
3. If `largeBlobKey` is not included in the authenticatorGetAssertion response structure (i.e., *not* in the extensions field of the authenticator data) then return that no large blob exists.
4. Let `key` be the value of `largeBlobKey` in the assertion result. If it is not 32 bytes long, return an error.
5. Fetch the large-blob array. If this fails, return an error.
6. For each element in that array:
    1. If the element is not a map conforming to the large-blob map structure defined above, skip this array element.
    2. Perform an AEAD_AES_256_GCM authenticated decryption of `ciphertext` using `key`, `nonce`, and the associated data specified above. If the decryption fails, skip this array element.
    3. Decompress the resulting plaintext with DEFLATE [RFC1951]. If decompression fails, return an error.
    4. If the length of the decompression result is not equal to `origSize`, return an error.
    5. Return the decompression result as the opaque large-blob data for the credential.
7. Return that no large blob exists.

> NOTE: DEFLATE has a maximum compression ratio of over 1000:1, thus the result of decompressing a small amount of data can be extremely large which might cause excessive memory use. Platforms SHOULD limit the maximum permitted value of `origSize` and that maximum SHOULD be at least 1MiB.

6.10.5. Writing per-credential large-blob data for a new credential

The platform SHOULD perform the following steps in order to write the opaque large-blob data for a new credential.

1. If the authenticator does not support the largeBlobKey extension, as defined in that section, return an error.
2. If the authenticatorMakeCredential operation for the new credential does not map `rk` to `true` in the `options` map, return an error. (Large blobs are only applicable for discoverable credentials.)
3. Perform the authenticatorMakeCredential operation for the new credential. In the `extensions` input additionally map `largeBlobKey` to `true`.
4. Let `key` be the largeBlobKey returned in the authenticatorMakeCredential response structure.
5. Let `origData` equal the opaque large-blob data.
6. Let `origSize` be the length, in bytes, of `origData`.
7. Let `plaintext` equal `origData` after compression with DEFLATE [RFC1951].
8. Let `nonce` be a fresh, random, 12-byte value.
9. Let `ciphertext` be the AEAD_AES_256_GCM authenticated encryption of `plaintext` using `key`, `nonce`, and the associated data as specified above.
10. Fetch the large-blob array. If this fails, return an error.
11. Append an element to the array, following the structure above, containing `nonce`, `origSize`, and `ciphertext`.
12. Perform the actions for writing the new large-blob array.

6.10.6. Updating per-credential large-blob data

Unlike the underlying largeBlobKey data, the opaque large-blob data for a credential may be updated or deleted. Given a credential, the platform SHOULD perform the following steps in order to update or delete it:

1. If the authenticator does not support the largeBlobKey extension, as defined in that section, return an error.
2. Perform an authenticatorGetAssertion operation with "largeBlobKey": `true` in the `extensions` map in order to fetch the largeBlobKey for the credential. (This step may be skipped if the pertinent output is already known.)
3. If `largeBlobKey` is not included in the authenticatorGetAssertion response structure (i.e., *not* in the extensions field of the authenticator data) then return that no large blob exists.
4. Let `key` be the value of `largeBlobKey` in the authenticatorGetAssertion response structure. If it is not 32 bytes long, return an error.
5. Fetch the large-blob array. If this fails, return an error.
6. For each element in that array:
    1. If the element is not a map conforming to the large-blob map structure defined above, skip this array element.
    2. Perform an AEAD_AES_256_GCM authenticated decryption of `ciphertext` using `key`, `nonce`, and the associated data specified above. If the decryption fails, skip this array element.
    3. If the platform wishes to delete the opaque large-blob data:
        1. Erase the current array element.
    4. Else (i.e. the platform wishes to update the opaque large-blob data):
        1. Let `origData` equal the new opaque large-blob data.
        2. Let `origSize` be the length, in bytes, of `origData`.
        3. Let `plaintext` equal `origData` after compression with DEFLATE [RFC1951].
        4. Let `nonce` be a fresh, random, 12-byte value.
        5. Let `ciphertext` be the AEAD_AES_256_GCM authenticated encryption of `plaintext` using `key`, `nonce`, and the associated data as specified above.
        6. Replace the current array element with a map, following the structure above, containing `nonce`, `origSize`, and `ciphertext`.
    5. Perform the actions for writing the new large-blob array.
    6. Return success.
7. Return an error.

6.10.7. Garbage collection of large-blob data

Large blobs may remain even when the linked credential has been erased. This can occur when a platform that doesn't support large blobs deletes a credential, or when a credential is implicitly deleted because a new credential with the same user ID and RP ID is created. Thus platform MAY perform a garbage collection at will and SHOULD perform a garbage collection when a large-blob cannot be stored because of lack of space, or when using credential management to enumerate credentials for other reasons.

Performing a garbage collection involves the following steps:

1. If `credMgmt` is not present in the `options` field of the authenticatorGetInfo response, garbage collection is not possible.
2. Use the authenticatorCredentialManagement command to enumerate all RPs with discoverable credentials, and then to enumerate all credentials for each of them.
3. Collect the set of `largeBlobKey` values returned, ignoring any that are not 32 bytes long.
4. Fetch the large-blob array. If this fails, return an error.
5. For each element in that array:
    1. If the element is not a map conforming to the large-blob map structure defined above, skip this array element. (The large-blob map is permitted to include extra elements.)
    2. Perform an AEAD_AES_256_GCM authenticated decryption of `ciphertext` using `nonce`, the associated data specified above, and each of the `largeBlobKey` values in turn as the key. If the decryption fails in every case, erase this array element.
6. If any array elements were erased then perform the actions for writing the updated large-blob array.

6.11. authenticatorConfig (0x0D)

NOTE: Platforms MUST NOT invoke this command unless the authnrCfg option ID is present and `true` in the response to an authenticatorGetInfo command.

This command is used to configure various authenticator features through the use of its subcommands.

It takes the following input map containing its input parameters:

Parameter name	Data type	Required?	Notes
subCommand (0x01)	Unsigned Integer	Required	subCommand currently being requested
subCommandParams (0x02)	CBOR Map	Optional	Map of subCommands parameters.
pinUvAuthProtocol (0x03)	Unsigned Integer	Optional	PIN/UV protocol version chosen by the platform.
pinUvAuthParam (0x04)	Byte String	Optional	The output of calling authenticate on some context specific to the subcommand.

The **currently defined authenticatorConfig subcommands** are:

subCommand Name	subCommand Number
enableEnterpriseAttestation	0x01
toggleAlwaysUv	0x02
setMinPINLength	0x03
vendorPrototype	0xFF

This authenticatorConfig command allows the platform to invoke various simple configuration operations on an authenticator. Parameters may be passed into subcommands, and only status codes are returned (i.e. no response map is defined). Typically, the platform may subsequently request and examine an authenticatorGetInfo response, per directions given for each subcommand, in order to ascertain results of having invoked the subcommand.

Authenticators MAY implement none, some, or all currently defined authenticatorConfig subcommands.

> NOTE: The vendorPrototype subCommand is reserved for vendor-specific authenticator configuration and experimentation. Platforms are not expected to generally utilize this subCommand.

To invoke `authenticatorConfig` the platform performs the following actions:

1. The platform sends the `authenticatorConfig` command with the following parameters:
    1. subCommand (0x01): The subcommand selected by the platform from the currently defined authenticatorConfig subcommands.
    2. subCommandParams (0x02): Map containing subcommand parameters, if the selected subcommand takes parameters.
    3. pinUvAuthProtocol (0x03): as selected when obtaining the shared secret.
    4. pinUvAuthParam (0x04): the result of calling `authenticate(pinUvAuthToken, 32×0xff || 0x0d || uint8(subCommand) || subCommandParams)`.

The authenticator performs the following actions upon receipt of this command:

1. If `subCommand` is not present in the input map, return CTAP2_ERR_MISSING_PARAMETER.
2. If the authenticator does not support the subcommand being invoked, per `subCommand`'s value, return CTAP1_ERR_INVALID_PARAMETER.
3. If the following statements are all true:
    1. `subCommand` value is toggleAlwaysUv (0x02).
    2. The authenticator is not protected by some form of user verification
    3. The alwaysUv option ID is present and `true`.

    then go to Step 5.

    > NOTE: This allows for initial configuration of authenticators that have the Always UV feature enabled by default.
4. If the authenticator is protected by some form of user verification or the alwaysUv option ID is present and `true`:
    1. If `pinUvAuthParam` is absent from the input map, then end the operation by returning CTAP2_ERR_PUAT_REQUIRED.
    2. If `pinUvAuthProtocol` is absent from the input map, then end the operation by returning CTAP2_ERR_MISSING_PARAMETER.

3. If `pinUvAuthProtocol` is not supported, return CTAP1_ERR_INVALID_PARAMETER.
4. Call `verify(pinUvAuthToken, 32×0xff || 0x0d || uint8(subCommand) || subCommandParams, pinUvAuthParam)`.
    1. If the verification fails, return CTAP2_ERR_PIN_AUTH_INVALID.
5. Check whether the pinUvAuthToken has the acfg permission. If not, return CTAP2_ERR_PIN_AUTH_INVALID.

5. Invoke `subCommand` (see below subsections for each defined subcommand), passing it the subCommandParams map.
6. Return the resulting status code as produced by `subCommand`, as defined in each subcommand subsection below.

> NOTE: User verification is only checked above if user verification is configured on a device. This implies that `authenticatorConfig` can be invoked without user verification if user verification is not configured, and the Always UV feature is disabled. This allows organisations to configure authenticators suitably for their environment before distributing them to users. See also authenticatorLargeBlobs.

6.11.1. Enable Enterprise Attestation

This `enableEnterpriseAttestation` subcommand is only implemented if the enterprise attestation feature is supported. This subcommand does not take any parameters: `subCommandParams` is ignored.

This subcommand performs the following steps:

1. If the enterprise attestation feature is disabled, then re-enable the enterprise attestation feature and return CTAP2_OK.

    > NOTE: Upon re-enabling the enterprise attestation feature, the authenticator will return an ep option id with the value of `true` in the authenticatorGetInfo command response upon receipt of subsequent authenticatorGetInfo commands.

2. Else (implying the enterprise attestation feature is enabled) take no action and return CTAP2_OK.

6.11.2. Toggle Always Require User Verification

This `toggleAlwaysUv` subcommand is only implemented if the Always Require User Verification feature is supported. This subcommand does not take any parameters: `subCommandParams` is ignored.

This subcommand performs the following steps:

1. If the alwaysUv feature is disabled:
    1. If the makeCredUvNotRqd option ID is present and `true`, then disable the makeCredUvNotRqd feature and set the makeCredUvNotRqd option ID to `false` or absent.
    2. Enable the alwaysUv feature and return CTAP2_OK.

    > NOTE: Upon enabling the Always Require User Verification feature, the authenticator will return an alwaysUv option ID with the value of `true` in the authenticatorGetInfo command response upon receipt of subsequent authenticatorGetInfo commands.

2. Else (implying the alwaysUv feature is enabled)
    1. If disabling the feature is supported:
        1. Set the makeCredUvNotRqd option ID to its default.
        2. Disable the alwaysUv feature and return CTAP2_OK.
    2. Else return CTAP2_ERR_OPERATION_DENIED.

    > NOTE: Authenticators SHOULD support users disabling the Always Require User Verification feature unless required not to by specific external certifications such as [CMVP].

6.11.3. Vendor Prototype Command

This subCommand allows vendors to test authenticator configuration features.

This `vendorPrototype` subcommand is only implemented if the vendorPrototypeConfigCommands member in the authenticatorGetInfo response is present.

Vendors SHOULD place implemented vendorCommandId values in the vendorPrototypeConfigCommands array.

subCommandParams Fields:

Field name	Data type	Required?	Definition
vendorCommandId (0x01)	Unsigned Integer	Required	Vendor-assigned command ID NOTE: If, and only if, this vendorCommandId (0x01) appears in this subCommandParams map and has a non-empty value, then other fields MAY also appear in the map, the map keys and associated values of which are vendor-defined.

This subCommand MUST include a subCommandParams map that MUST contain vendorCommandId as a member. The vendor randomly selects a 64-bit Unsigned Integer value to use for the value of vendorCommandId, e.g., by using a cryptographic random number generator. An *example* of such a vendorCommandId value is (in hex): `0x4e5a15aa89d2b8b6`. This approach avoids collisions amongst different vendors' vendorCommandIds. Thus there is no need for a registry of vendorCommandId values. One way to easily generate such values is by using the commonly available openssl tool.

This subCommand performs the following steps:

1. If the vendorCommandId value is unknown:
    1. return CTAP2_ERR_INVALID_SUBCOMMAND
2. Else: (implying the vendorCommandId value is known)
    1. Extract any additional members form the subCommandParams map.
    2. Perform Vendor Command specific processing and return any status code it generates. Success MUST be indicated by returning CTAP2_OK.

NOTE: Vendors MUST NOT count on obscurity of the vendorCommandId value as any sort of security.

6.11.4. Setting a minimum PIN Length

This `setMinPINLength` subcommand is only implemented if the setMinPINLength option ID is present.

This command sets the minimum PIN length in Unicode code points to be enforced by the authenticator while changing/setting up a ClientPIN.

NOTE: This is not applicable for any other type of PIN functionality the authenticator may have.

`subCommandParams` members defined for this subcommand:

Parameter name	Data type	Required?	Definition
newMinPINLength (0x01)	Unsigned Integer	Optional	Minimum PIN length in code points
minPinLengthRPIDs (0x02)	Array of strings	Optional	RP IDs which are allowed to get this information via the minPinLength extension. This parameter MUST NOT be used unless the minPinLength extension is supported.
forceChangePin (0x03)	Boolean	Optional	The authenticator returns CTAP2_ERR_PIN_POLICY_VIOLATION until changePIN is successful.
pinComplexityPolicy (0x04)	Boolean	Optional	If set to TRUE the authenticator enforces a PIN complexity policy until the authenticator is reset.

1. Platform sends the following subCommandParams (0x03) map containing following parameters:
    1. newMinPINLength (0x01) (Optional): Minimum PIN length in code points
    2. minPinLengthRPIDs (0x02) (Optional): List of RP IDs allowed to get the current newMinPINLength via minPinLength extension.
    3. forceChangePin (0x03) (Optional): If `true` a PIN change is required after this command.
    4. pinComplexityPolicy (0x04) (Optional): If `true` a PIN complexity policy is enforced after this command.

2. Authenticator performs following operations upon receiving the request:
    1. If newMinPINLength is absent, then let newMinPINLength be present with the value of current minimum PIN length.
    2. If minPinLengthRPIDs is present and the authenticator does not support the minPinLength extension, return CTAP1_ERR_INVALID_PARAMETER.
    3. If newMinPINLength is less than the current minimum PIN length, return CTAP2_ERR_PIN_POLICY_VIOLATION.

        NOTE: Minimum PIN lengths may only be increased; they cannot be made shorter.

        NOTE: The authenticator must be reset to return the current minimum PIN length to the pre-configured minimum PIN length.
    4. If the value of forceChangePin is `true`, then:
        1. If the value of clientPIN is `false`, then return CTAP2_ERR_PIN_NOT_SET.
        2. Let the value of the forcePINChange member of the authenticatorGetInfo response be `true`.

            NOTE: This will force the user to change their PIN upon the next use of the authenticator, if a PIN is set.
    5. If the value of pinComplexityPolicy is `true`, then:
        1. Let the value of the pinComplexityPolicy authenticatorGetInfo response member be `true`.
    6. If the value of PINCodePointLength is less than newMinPINLength and the value of clientPIN is `true` then let the value of the forcePINChange member of the authenticatorGetInfo response be `true`.
    7. If the value of the forcePINChange member of the authenticatorGetInfo response is `true`, then:
        1. Authenticator calls resetPersistentPinUvAuthToken() (all persistent permissions are cleared on pin change).
        2. Authenticator calls resetPinUvAuthToken() for *all* pinUvAuthProtocols supported by this authenticator. (I.e. all existing pinUvAuthTokens are invalidated.)
    8. If minPinLengthRPIDs is present and contains at least one string, then:
        1. Platform can track how many RP IDs it can set, by checking value of the maxRPIDsForSetMinPINLength member of the authenticatorGetInfo. If the supplied list larger than the maxRPIDsForSetMinPINLength, then authenticator must return an error.
        2. If the authenticator does not have a pre-configured list of RP IDs authorized to receive the current minimum PIN length value, the authenticator stores the minPinLengthRPIDs parameter's list as the entire list of RP IDs authorized to receive the current minimum PIN length value.
        3. Otherwise, if the authenticator has a pre-configured list of RP IDs authorized to receive the current minimum PIN length value, it adds the minPinLengthRPIDs parameter's list to the immutable pre-configured list. Any previously added RP IDs are overwritten.

            NOTE: How the authenticator "adds" the minPinLengthRPIDs parameter's list to the pre-configured list is an implementation detail.
        4. If the authenticator cannot store or add the minPinLengthRPIDs, it returns CTAP2_ERR_KEY_STORE_FULL.
    9. Authenticator returns CTAP2_OK.

6.12. Prototype authenticatorBioEnrollment (0x40) (For backwards compatibility with "FIDO_2_1_PRE")

This superseded command is OPTIONAL and ONLY provided for backwards compatibility with platforms that implemented "FIDO_2_1_PRE" functionality, and have not been updated to "FIDO_2_1 or FIDO_2_2". CTAP2.1 platforms MUST NOT use this command if bioEnroll option ID is present in the authenticatorGetInfo response.

If a CTAP2.1 authenticator implements this prototype (0x40) command:

1. The authenticator MUST also implement the authenticatorBioEnrollment (0x09) commands.
2. The authenticator MUST provide the bioEnroll option ID in the authenticatorGetInfo response for feature detection of the CTAP2.1 feature.
3. The authenticator MUST utilize the appropriate PIN protocol's verify() function to validate the `pinUvAuthParam` (referred to as `pinAuth` in the Bio Enrollment Prototype specification), and MUST return CTAP2_ERR_PIN_AUTH_INVALID if verify() returns `error`.

The feature detection logic for the Bio Enrollment Prototype vendor specific feature is:

1. "FIDO_2_1_PRE" is present in the authenticatorGetInfo response versions member.
2. The userVerificationMgmtPreview option ID in the authenticatorGetInfo response is present and `true`.

This preview command does not require permissions, thus it is compatible with a pinUvAuthToken generated by the getPinToken command. CTAP 2.1 platforms MUST use the newer authenticatorBioEnrollment (0x09) command if the authenticator supports it.

6.13. Prototype authenticatorCredentialManagement (0x41) (For backwards compatibility with "FIDO_2_1_PRE")

This superseded command is OPTIONAL and ONLY provided for backwards compatibility with platforms that implemented "FIDO_2_1_PRE" functionality, and have not been updated to "FIDO_2_1 or FIDO_2_2". CTAP2.1 platforms MUST NOT use this command if credMgmt option ID is present in the authenticatorGetInfo response.

If a CTAP2.1 authenticator implements this prototype (0x41) command:

1. The authenticator MUST also implement the authenticatorCredentialManagement (0x0A) commands.
2. The authenticator MUST provide the credMgmt option ID in the authenticatorGetInfo response for feature detection of the CTAP2.1 feature.
3. The authenticator MUST utilize the appropriate PIN protocol's verify() function to validate the `pinUvAuthParam` (referred to as `pinAuth` in the Credential Management Prototype specification), and MUST return CTAP2_ERR_PIN_AUTH_INVALID if verify() returns `error`.

The feature detection logic for the Credential Management Prototype vendor specific feature is:

1. "FIDO_2_1_PRE" is present in the authenticatorGetInfo response versions member.
2. The credentialMgmtPreview option ID in the authenticatorGetInfo response is present and `true`.

This preview command does not require permissions, thus it is compatible with a pinUvAuthToken generated by the getPinToken command. CTAP 2.1 platforms MUST use the newer authenticatorCredentialManagement (0x0A) command if the authenticator supports it.

7. Feature-Specific Descriptions and Actions

This section provides detailed descriptions of specific features along with normative feature-specific platform (and possibly authenticator) actions whose specification is not appropriate to include in other parts of this specification.

7.1. Enterprise Attestation

An **enterprise** is some form of organization, often a business entity. An **enterprise context** is in effect when a device, e.g., a computer, an authenticator, etc., is controlled by an enterprise.

An enterprise attestation is an attestation 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.

The expectation is that enterprises will work directly with their authenticator vendor(s) in order to source their enterprise attestation capable authenticators.

An enterprise attestation capable authenticator MAY be configured to support either or both:

- **Vendor-facilitated enterprise attestation**:

  In this case, an enterprise attestation capable authenticator, on which enterprise attestation is enabled, upon receiving the `enterpriseAttestation` parameter with a value of 1 (or 2, see Note below) on a `authenticatorMakeCredential` command, will provide enterprise attestation to a non-updateable **pre-configured RP ID list**, as identified by the enterprise and provided to the authenticator vendor, which is "burned into" the authenticator by the vendor.

  If enterprise attestation is requested for any RP ID other than the pre-configured RP ID(s), the attestation returned along with the new credential is a regular privacy-preserving attestation, i.e., NOT an enterprise attestation.

- **Platform-managed enterprise attestation**:

  In this case, an enterprise attestation capable authenticator on which enterprise attestation is enabled, upon receiving the `enterpriseAttestation` parameter with a value of 2 on a `authenticatorMakeCredential` command, will return an enterprise attestation. The platform is enterprise-managed and has already performed the necessary vetting of the RP ID.

  > NOTE: Authenticators wishing to support only vendor-facilitated enterprise attestation MAY treat `enterpriseAttestation = 2` the same as `enterpriseAttestation = 1`.

7.1.1. Feature detection

The ep option ID in the authenticatorGetInfo response defines feature support detection for this feature.

7.1.2. Platform Actions

A platform wishing to obtain an enterprise attestation, e.g., when running in an enterprise context, SHOULD invoke the `authenticatorMakeCredential` operation in the following manner:

1. Invoke the `authenticatorGetInfo` command and examine the returned response structure for the ep Option ID. If ep is not present or present and set to false, the platform SHOULD either terminate these steps or invoke the `authenticatorMakeCredential` command without the `enterpriseAttestation` parameter, and skip the following steps.
2. Invoke the `authenticatorMakeCredential` command and pass the `enterpriseAttestation` parameter with a value of either 1 or 2.
3. If the platform is operating in a non-enterprise context, it SHOULD display an explicit warning to the user, including the RP ID, notifying the user that they are being uniquely identified to this Relying Party.

7.1.3. Authenticator Actions

If an enterprise attestation capable authenticator receives an authenticatorReset command, it MUST disable the enterprise attestation feature. The enterprise attestation feature may be re-enabled by invoking the authenticatorConfig command's enable-enterprise-attestation subcommand.

7.2. Always Require User Verification

This feature allows a user to protect the credentials on their authenticator with some form of user verification independent of the Relying Party requesting some form of user verification in its higher-level API request, e.g., via [WebAuthn]. Platform authenticators and other authenticators with the alwaysUv feature enabled will always perform user verification and set the "uv" bit to `true` in the response, e.g., even if the Relying Party sets user verification to `Discouraged` in a [WebAuthn] request. Some external certification programs such as [CMVP] for [FIPS140-3] prohibit the authenticator performing signing operations without authentication. This feature allows authenticators to conform to such non FIDO certification requirements.

> NOTE: Platform authenticators typically provide users and platforms this sort of behaviour via private API.

7.2.1. Feature detection

The alwaysUv option ID in the authenticatorGetInfo response defines feature support detection for this feature.

7.2.2. Platform Actions

1. If the feature is supported and enabled: (alwaysUv is present and `true`)
    1. The platform SHOULD treat all Relying Party requests (e.g., those being made by a Relying Party via [WebAuthn] or a platform API) as requiring user verification.
    2. If the authenticator is not protected by some form of user verification, the platforms SHOULD help users enroll a clientPin and or a built-in user verification method, if either or both are supported.
2. Platforms may enable or disable this feature by invoking the authenticatorConfig command's toggleAlwaysUv subcommand.

7.2.3. Authenticator Actions

1. If the feature is supported and enabled: (alwaysUv is present and `true`)
    1. The authenticator MUST require some form of user verification for the authenticatorMakeCredential and authenticatorGetAssertion commands.
    2. Authenticators supporting CTAP1/U2F MUST protect the credentials with built-in user verification methods, or disable CTAP1/U2F when the alwaysUv option ID is present and `true`.
    3. If the "uv" bit set in the response is `false` some authenticators conforming to [FIPS140-3] or other security requirements may return an syntactically-correct but invalid signature (i.e., one that no credential public key minted by this authenticator, now or ever, will match) rather than a signature from the private key from the selected credential. An example for a ECDSA signature is to return a fixed value of (1, 1). Thus the returned signature will not be verifiable, which is up to the Relying Party to handle. This approach avoids returning an error to the platform because doing that would interfere with some platforms' approach of "pre-flighting" the allowList or excludeList.
2. If the feature is supported and disabled: (alwaysUv is present and `false`)
    1. The authenticator does not always require user verification for its operations. It is dependent on the parameters passed to individual operations as specified herein.

3. After an authenticator reset:
    1. Set the makeCredUvNotRqd option ID to its default pre-configured state.
    2. Set the alwaysUv option ID to its default pre-configured state (may be either `true` or `false`).

7.2.4. Disabling CTAP1/U2F

Authenticators MUST disable CTAP1/U2F when the alwaysUv option ID is present and `true` in the authenticatorGetInfo response, unless the CTAP1/U2F authenticator is protected by a built-in user verification method. When CTAP1/U2F is disabled:

1. The authenticator MUST NOT return "U2F_V2" in the versions array.
2. The U2F_REGISTER and U2F_AUTHENTICATE commands MUST immediately fail and return SW_COMMAND_NOT_ALLOWED.

7.3. Authenticator Certifications

The certifications member provides a hint to the platform with additional information about certifications that the authenticator has received. Certification programs may revoke certification of specific devices at any time. Relying partys are responsible for validating attestations and AAGUID via appropriate methods. Platforms may alter their behaviour based on these hints such as selecting a PIN protocol or credProtect level.

7.3.1. Authenticator Actions

An authenticator's **supported certifications** MAY be returned in the certifications member of an authenticatorGetInfo response.

All certifications are in the form key-value pairs with string IDs and integer values. The following table lists all defined certification types as of CTAP version "FIDO_2_2":

certification ID	Definition
FIPS-CMVP-2	The [FIPS140-2] Cryptographic-Module-Validation-Program overall certification level. This is a integer from 1 to 4.
FIPS-CMVP-3	The [FIPS140-3] [CMVP] or ISO/IEC 19790:2012(E) and ISO/IEC 24759:2017(E) overall certification level. This is a integer from 1 to 4.
FIPS-CMVP-2-PHY	The [FIPS140-2] Cryptographic-Module-Validation-Program physical certification level. This is a integer from 1 to 4.
FIPS-CMVP-3-PHY	The [FIPS140-3] [CMVP] or ISO/IEC 19790:2012(E) and ISO/IEC 24759:2017(E) physical certification level. This is a integer from 1 to 4.
CC-EAL	Common Criteria Evaluation Assurance Level [CC1V3-1R5]. This is a integer from 1 to 7. The intermediate-plus levels are not represented.
FIDO	FIDO Alliance certification level. This is an integer from 1 to 6. The numbered levels are mapped to the odd numbers, with the plus levels mapped to the even numbers e.g., level 3+ is mapped to 6.

7.4. Set Minimum PIN Length

This feature allows a Relying Party (e.g., an enterprise) to enforce a minimum pin length policy for authenticators registering credentials by examining the return value of the Minimum PIN Length Extension (minPinLength). The authenticatorConfig command's setMinPINLength subCommand allows the platform to set the minimum pin length policy for authenticator, force a change of PIN before allowing User Verification, and setting the list of minPinLengthRPIDs that allow the specified RP ID to receive the extension response.

If this feature is supported, the authenticator MUST implement:

1. The ClientPIN feature.
2. The setMinPINLength subCommand of the authenticatorConfig command.
3. The Minimum PIN Length Extension (minPinLength).

7.4.1. Feature detection

The setMinPinLength option ID in the authenticatorGetInfo response defines feature support detection for this feature.

7.4.2. Platform Actions

NOTE: Because ClientPIN must be implemented for this set minimum PIN length feature to be implemented, basic minimum PIN length enforcement already occurs. This feature is only about providing for the minimum PIN length to be altered from its pre-configured value.

1. If the forcePINChange member of the authenticatorGetInfo response is present and `true`:
    1. The platform should guide the user to change the PIN before invoking the getPinToken or getPinUvAuthTokenUsingPinWithPermissions subcommands.
2. Platforms may perform the following actions by invoking the authenticatorConfig command's setMinPINLength subcommand:
    1. Increase the minimum pin length for clientPin.
    2. Set the minPinLengthRPIDs parameter's list to allow Relying Parties receiving the minPinLength extension.
    3. Set the authenticator to require a PIN change before allowing clientPin based authentication.
    4. Enable enforcement of a PIN complexity policy.

7.4.3. Authenticator Actions

1. If this feature is enabled the extension identifier minpinlength in the extensions member of the authenticatorGetInfo response MUST be present.
2. After an authenticator reset:
    1. Set the minPINLength member of the authenticatorGetInfo response to its default pre-configured minimum PIN length.
    2. Set the minPinLengthRPIDs parameter's list to the immutable pre-configured list, if any. Any previously added RP IDs are removed.
    3. Set the forcePINChange member of the authenticatorGetInfo response to `false`.

7.5. Set PIN Complexity Policy

This feature allows a Relying Party (e.g., an enterprise) to enforce a pin complexity policy for authenticators registering credentials by examining the return value of the PIN Complexity Policy Extension (pinComplexityPolicy). The authenticatorConfig command's setMinPINLength subCommand allows the platform to enable the PIN Complexity Policy for the authenticator, force a change of PIN before allowing User Verification, and setting the list of minPinLengthRPIDs that allow the specified RP ID to receive the extension response.

If this feature is supported, the authenticator MUST implement:

1. The ClientPIN feature.
2. The setMinPINLength subCommand of the authenticatorConfig command.
3. The Minimum PIN Length Extension (minPinLength).

7.5.1. Feature detection

The pinComplexityPolicy option ID in the authenticatorGetInfo response defines feature support detection for this feature.

7.5.2. Platform Actions

NOTE: Because ClientPIN must be implemented for this set PIN complexity policy feature to be implemented, basic minimum PIN length enforcement already occurs. This feature is only about providing for the minimum PIN length to be altered from its pre-configured value.

1. If the forcePINChange member of the authenticatorGetInfo response is present and `true`:
    1. The platform should guide the user to change the PIN before invoking the getPinToken or getPinUvAuthTokenUsingPinWithPermissions subcommands.
2. Platforms may perform the following actions by invoking the authenticatorConfig command's setMinPINLength subcommand:
    1. Increase the minimum pin length for clientPin.

2. Set the minPinLengthRPIDs parameter's list to allow Relying Parties receiving the minPinLength extension.
3. Set the authenticator to require a PIN change before allowing clientPin based authentication.
4. Enable enforcement of a PIN complexity policy.

7.5.3. Authenticator Actions

1. If this feature is enabled the extension identifier pinComplexityPolicy in the extensions member of the authenticatorGetInfo response MUST be present.
2. After an authenticator reset:
    1. Set the pinComplexityPolicy member of the authenticatorGetInfo response to its default pre-configured PIN complexity policy value, if any.
    2. Set the minPinLengthRPIDs parameter's list to the immutable pre-configured list, if any. Any previously added RP IDs are removed.
    3. Set the forcePINChange member of the authenticatorGetInfo response to `false`.

7.6. JSON-based Messages

7.6.1. Feature detection

Support for JSON-based messages, and more specifically Digital Credentials API requests using JSON-based messages, is determined by the presence of the string `dc` in the array for key 3 of the post handshake message's CBOR map as defined in Hybrid Transports.

7.6.2. Request Properties

The following JSON schema defines the properties of a JSON-based request:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://schemas.fidoalliance.org/ctap/json-request/v2_2rd02.schema.json",
  "title": "JSON-based Request",
  "type": "object",
  "properties": {
    "origin": {
      "type": "string",
      "description": "The caller's origin as determined by the [=client platform=]."
    },
    "requestType": {
      "type": "string",
      "description": "The type of request.",
      "anyOf": [
        {
          "const": "credential.get",
          "description": "A get request from Credential Management or the app platform equivale
nt."
        },
        {
          "const": "credential.create",
          "description": "A create request from Credential Management or the app platform equiv
alent."
        }
      ]
    },
    "request": {
      "type": "object",
      "description": "One or more requests of the same requestType.",
      "properties": {
        "digital": {
          "type": "object",
          "description": "The [=Digital Credentials API=] request object."
        }
      }
    }
  },
  "additionalProperties": false,
  "required": [
    "origin",
    "requestType",
    "request"
  ]
}
```

EXAMPLE 1
JSON-based request for a digital credential

```
{
  "origin": "https://verify1.example.com",
  "requestType": "credential.get",
  "request": {
    "digital": {
      "response_type": "vp_token",
      "nonce": "n-0S6_WzA2Mj",
      "client_metadata": {
        "jwks": {
          "keys": [
            {
              "kty": "EC",
              "crv": "P-256",
              "x": "MKBCTNIcKUSDii11ySs3526iDZ8AiTo7Tu6KPAqv7D4",
              "y": "4Etl6SRW2YiLUrN5vfvVHuhp7x8PxltmWWlbbM4IFyM",
              "use": "enc",
              "kid": "1"
            }
          ]
        }
      },
      "presentation_definition": {
        "id": "mDL-Request",
        "input_descriptors": [
          {
            "id": "org.iso.18013.5.1.mDL",
            "format": {
              "mso_mdoc": {
                "alg": [
                  "EdDSA",
                  "ES256"
                ]
              }
            },
            "constraints": {
              "limit_disclosure": "required",
              "fields": [
                {
                  "path": [
                    "$['org.iso.18013.5.1']['family_name']"
                  ],
                  "intent_to_retain": false
                },
                {
                  "path": [
                    "$['org.iso.18013.5.1']['portrait']"
                  ],
                  "intent_to_retain": false
                },
                {
                  "path": [
                    "$['org.iso.18013.5.1']['driving_privileges']"
                  ],
                  "intent_to_retain": false
                },
                {
                  "path": [
                    "$['domestic_namespace']['domestic_data_element_id']"
                  ],
                  "intent_to_retain": false
                }
              ]
            }
          }
        ]
      }
    }
  }
}
```

7.6.3. Response Properties

The following JSON schema defines the properties of a JSON-based response:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://schemas.fidoalliance.org/ctap/json-response/v2_2rd02.schema.json",
  "title": "JSON-based Response",
  "type": "object",
  "properties": {
    "response": {
      "type": "object",
      "description": "One or more responses matching the request.",
      "properties": {
        "digital": {
          "type": "object",
          "description": "A response to a Digital Credential request.",
          "maxProperties": 1,
          "properties": {
            "data": {
              "type": "object",
              "description": "The [=Digital Credentials API=] response object."
            },
            "error": {
              "type": "string",
              "description": "For an unsuccessful ceremony, the error code describing the error
dition.",
              "anyOf": [
                {
                  "const": "USER_CANCELLED",
                  "description": "The user actively cancelled the request."
                },
                {
                  "const": "DEVICE_ABORTED",
                  "description": "The device aborted the request."
                },
                {
                  "const": "NO_CREDENTIAL",
                  "description": "No credential found to satisfy the request."
                }
              ]
            },
            "additionalProperties": false
          }
        }
      }
    }
  },
  "additionalProperties": false,
  "required": [
    "response"
  ]
}
```

EXAMPLE 2

Successful JSON-based response for a digital credential:


```
{
  "response": {
    "digital": {
      "data": {
        "protocol": "openid4vp",
        "data": {
          "presentation_submission": {
            "definition_id": "example_jwt_vc",
            "id": "example_jwt_vc_presentation_submission",
            "descriptor_map": [
              {
                "id": "id_credential",
                "path": "$",
                "format": "jwt_vp_json",
                "path_nested": {
                  "path": "$.vp.verifiableCredential[0]",
                  "format": "jwt_vc_json"
                }
              }
            ]
          },
          "vp_token": "eyJhbGciOiAiRVMyNTYiLCAidHlwIjogInZjK3NkLWp3dCIsICJraWQiOiAiZG9jLXNp
Z25lci0wNS0yNS0yMDIyIn0.eyJfc2QiOiBbIjNvVUNuYUt0N3dxREt1eWgtTGdRb3p6ZmhnYjhnTzVOaS1SQ1dzV1c
dkEiLCAiOHo4ejlYOWpVdGI5OWdqZWpDd0ZBR3o0YXFsSGYtc0NxUTZlTV9xbXBVUSIsICJDeHE0ODcyVVhYbmdHVU
X2tsOGZkd1ZGa3lLNkFKZlBaTHk3TDVfMGtJIiwgIlRHZjRvTGJnd2Q1SlFhSHlLVlFaVTlVZEdFMHc1cnREc3Jaem
YW9tTG8iLCAianN1OXlWdWx3UVFsaEZsTV8zSmx6TWFTRnpnbGhRRzBEcGZheVF3TFVLNCIsICJzRmNWaUhOLUpHM2
VXlCbVU0Zmt3dXN5NUkxU0xCaGUxak52S3hQNXhNIiwgInRpVG5ncDlfamhDMzg5VVA4X2s2N01YcW9TZmlIcTNpSz
OXVuNHdlX1kiLCAieHNLa0dKWEQxLWUzSTl6ajBZeUtOdi1sVTVZcWhzRUFGOU5oT3I4eGdhNCJdLCAiaXNzIjogImh
dHBzOi8vZXhhbXBsZS5jb20vaXNzdWVyIiwgImlhdCI6IDE2ODMwMDAwMDAsICJleHAiOiAxODgzMDAwMDAwLCAidmV
IjogImh0dHBzOi8vY3JlZGVudGlhbHMuZXhhbXBsZS5jb20vaWRlbnRpdHlfY3JlZGVudGlhbCIsICJfc2RfYWxnIjo
InNoYS0yNTYiLCAiY25mIjogeyJqd2siOiB7Imt0eSI6ICJFQyIsICJjcnYiOiAiUC0yNTYiLCAieCI6ICJUQ0FFUjE
WnZ1M09IRjRqNFc0dmZTVm9ISVAxSUxpbERsczd2Q2VHZW1jIiwgInkiOiAiWnhqaVdXYlpNUUdIVldLVlE0aGJTSWl
c1ZmdWVjQ0U2dDRqVDlGMkhaUSJ9fX0.hBeB-fuMsIQ82QIE_674CSPIufs7w0D9CdfGdP_tGyBVp_vTSlbWb9MInFk
Z6Y3ie-r0MMeSSEHyuUz9WNGSQ~WyJlbHVWNU9nM2dTTklJOEVZbnN4QV9BIiwgImZhbWlseV9uYW1lIiwgIkRvZSJd
WyIyR0xDNDJzS1F2ZUNmR2ZyeU5STjl3IiwgImdpdmVuX25hbWUiLCAiSm9obiJd~eyJhbGciOiAiRVMyNTYiLCAidH
wIjogImtiK2p3dCJ9.eyJub25jZSI6ICJuLTBTNl9XekEyTWoiLCAiYXVkIjogImh0dHBzOi8vZXhhbXBsZS5jb20vc
VyaWZpZXIiLCAiaWF0IjogMTcwOTgzODYwNCwgInNkX2hhc2giOiAiRHktUll3WmZhYW9DM2luSmJMc2xnUHZNcDA5Y
gtY2xZUF8zcWJScXRXNCJ9.RmgIhqCHYWerxbDboMuB0lli63HPJHI9Vl2ZNOGh20C7_6p7nf3Wkd2wkx5WlmwTwtH
87MBY2nuRLoeduQMA"
        }
      }
    }
  }
}
```


EXAMPLE 3

Unsuccessful JSON-based response for a digital credential


```
{
  "response": {
    "digital": {
      "error": "NO_CREDENTIAL"
    }
  }
}
```


8. Message Encoding

Many transports (e.g., Bluetooth Smart) are bandwidth-constrained, and serialization formats such as JSON are too heavy-weight for such environments. For this reason, all encoding is done using the concise binary encoding CBOR [RFC8949].

To reduce the complexity of the messages and the resources required to parse and validate them, all messages MUST use the CTAP2 canonical CBOR encoding form as specified below, which differs from the "Deterministically Encoded CBOR" suggested in Section 4.2 of [RFC8949]. All encoders MUST serialize CBOR in the CTAP2 canonical CBOR encoding form without duplicate map keys. All decoders SHOULD reject CBOR that is not validly encoded in the CTAP2 canonical CBOR encoding form and SHOULD reject messages with duplicate map keys.

The **CTAP2 canonical CBOR encoding form** uses the following rules:

- Integers MUST be encoded as small as possible.
    - 0 to 23 and -1 to -24 MUST be expressed in the same byte as the major type;

- 24 to 255 and -25 to -256 MUST be expressed only with an additional uint8_t;
- 256 to 65535 and -257 to -65536 MUST be expressed only with an additional uint16_t;
- 65536 to 4294967295 and -65537 to -4294967296 MUST be expressed only with an additional uint32_t.

- The representations of any floating-point values are not changed.

> NOTE: The size of a floating point value—16-, 32-, or 64-bits—is considered part of the value for the purpose of CTAP2. E.g., a 16-bit value of 1.5, say, has different semantic meaning than a 32-bit value of 1.5, and both can be canonical for their own meanings.

- The expression of lengths in major types 2 through 5 MUST be as short as possible. The rules for these lengths follow the above rule for integers.
- Indefinite-length items MUST be made into definite-length items.
- The keys in every map MUST be sorted lowest value to highest. The sorting rules are:
  - If the major types are different, the one with the lower value in numerical order sorts earlier.
  - If two keys have different lengths, the shorter one sorts earlier;
  - If two keys have the same length, the one with the lower value in (byte-wise) lexical order sorts earlier.

> NOTE: These rules are equivalent to a lexicographical comparison of the canonical encoding of keys for major types 0-3 and 7 (integers, strings, and simple values). They differ for major types 4-6 (arrays, maps, and tags), which CTAP2 does not use as keys in maps. These rules should be revisited if CTAP2 does start using the complex major types as keys.

- Tags as defined in Section 3.4 in[RFC8949] MUST NOT be present.

Because some authenticators are memory constrained, the depth of nested CBOR structures used by all message encodings is limited to at most four (4) levels of any combination of CBOR maps and/or CBOR arrays. Authenticators MUST support at least 4 levels of CBOR nesting. Clients, platforms, and servers MUST NOT use more than 4 levels of CBOR nesting.

Likewise, because some authenticators are memory constrained, the maximum message size supported by an authenticator MAY be limited. By default, authenticators MUST support messages of at least 1024 bytes. Authenticators MAY declare a different maximum message size supported using the maxMsgSize authenticatorGetInfo result parameter. Clients, platforms, and servers MUST NOT send messages larger than 1024 bytes unless the authenticator's maxMsgSize indicates support for the larger message size. Authenticators MAY return the CTAP2_ERR_REQUEST_TOO_LARGE error if size or memory constraints are exceeded.

If map keys are present that an implementation does not understand, they MUST be ignored. Note that this enables additional fields to be used as new features are added without breaking existing implementations.

Messages from the host to authenticator are called "commands" and messages from authenticator to host are called "responses". All values are big endian encoded.

Authenticators SHOULD return the CTAP2_ERR_INVALID_CBOR error if received CBOR does not conform to the requirements above.

Several commands reference externally-defined structures such as`PublicKeyCredentialRpEntity` which, for the purposes of this protocol, are encoded as CBOR. The rules and behaviours for processing such CBOR are defined above, but such structures can also be invalid because of missing required fields, or because values have an incorrect type. If structures in messages from the host are missing required members, or the values of those members have the wrong type, then the authenticator SHOULD return CTAP2_ERR_CBOR_UNEXPECTED_TYPE.

8.1. Command Codes§

The assigned values for vendor specific commands and their descriptions are:

Command Name	Command Code	Has parameters?
authenticatorVendorFirst	0x40	NA
Vendor - Bio Enrollment Prototype	0x40	yes
Vendor - Credential Management Prototype	0x41	yes
authenticatorVendorLast	0xBF	NA

If an authenticator receives a command code it does not implement, it MUST return CTAP1_ERR_INVALID_COMMAND. If the authenticator implements a command code having subcommands, but does not implement an invoked subcommand, it MUST return CTAP2_ERR_INVALID_SUBCOMMAND.

> NOTE: Some authenticators implementing earlier versions of this specification may not behave as specified by the prior paragraph, because this behavior was only implied at that time.

Command codes in the range between **authenticatorVendorFirst** and **authenticatorVendorLast** may be used for vendor-specific implementations. For example, the vendor may choose to put in some testing commands. Note that the FIDO client will never generate these commands. All other command codes are reserved for future use and may not be used.

Command parameters are encoded using a CBOR map (CBOR major type 5). The CBOR map MUST be encoded using the definite length variant.

Some commands have optional parameters. Therefore, the length of the parameter map for these commands may vary. For example, authenticatorMakeCredential may have 4, 5, 6, or 7 parameters, while authenticatorGetAssertion may have 2, 3, 4, or 5 parameters.

All command parameters are CBOR encoded following the *JSON to CBOR* conversion procedures as per the CBOR specification [RFC8949]. Specifically, parameters that are represented as DOM objects in the *Authenticator API* layers (formally defined in the Web API[WebAuthn]) are converted first to JSON and subsequently to CBOR.

8.2. Status codes§

The error response values range from 0x01 - 0xff. This range is split based on error type.

Error response values in the range between **CTAP2_OK** and **CTAP2_ERR_SPEC_LAST** are reserved for spec purposes.

Error response values in the range between **CTAP2_ERR_VENDOR_FIRST** and **CTAP2_ERR_VENDOR_LAST** may be used for vendor-specific implementations. All other response values are reserved for future use and may not be used. These vendor specific error codes are not interoperable and the platform SHOULD treat these errors as any other unknown error codes.

Error response values in the range between **CTAP2_ERR_EXTENSION_FIRST** and **CTAP2_ERR_EXTENSION_LAST** may be used for extension-specific implementations. These errors need to be interoperable for vendors who decide to implement such optional extension.

Code	Name	Description
0x00	CTAP1_ERR_SUCCESS, CTAP2_OK	Indicates successful response.
0x01	CTAP1_ERR_INVALID_COMMAND	The command is not a valid CTAP command.
0x02	CTAP1_ERR_INVALID_PARAMETER	The command included an invalid parameter.
0x03	CTAP1_ERR_INVALID_LENGTH	Invalid message or item length.
0x04	CTAP1_ERR_INVALID_SEQ	Invalid message sequencing.
0x05	CTAP1_ERR_TIMEOUT	Message timed out.
0x06	CTAP1_ERR_CHANNEL_BUSY	Channel busy. Client SHOULD retry the request after a short delay. Note that the client MAY abort the transaction if the command is no longer relevant.
0x0A	CTAP1_ERR_LOCK_REQUIRED	Command requires channel lock.
0x0B	CTAP1_ERR_INVALID_CHANNEL	Command not allowed on this cid.
0x11	CTAP2_ERR_CBOR_UNEXPECTED_TYPE	Invalid/unexpected CBOR error.
0x12	CTAP2_ERR_INVALID_CBOR	Error when parsing CBOR.
0x14	CTAP2_ERR_MISSING_PARAMETER	Missing non-optional parameter.
0x15	CTAP2_ERR_LIMIT_EXCEEDED	Limit for number of items exceeded.
0x17	CTAP2_ERR_FP_DATABASE_FULL	Fingerprint data base is full, e.g., during enrollment.
0x18	CTAP2_ERR_LARGE_BLOB_STORAGE_FULL	Large blob storage is full. (See § 6.10.3 Large, per-credential blobs.)
0x19	CTAP2_ERR_CREDENTIAL_EXCLUDED	Valid credential found in the exclude list.
0x21	CTAP2_ERR_PROCESSING	Processing (Lengthy operation is in progress).
0x22	CTAP2_ERR_INVALID_CREDENTIAL	Credential not valid for the authenticator.
0x23	CTAP2_ERR_USER_ACTION_PENDING	Authentication is waiting for user interaction.

Code	Name	Description
0x24	CTAP2_ERR_OPERATION_PENDING	Processing, lengthy operation is in progress.
0x25	CTAP2_ERR_NO_OPERATIONS	No request is pending.
0x26	CTAP2_ERR_UNSUPPORTED_ALGORITHM	Authenticator does not support requested algorithm.
0x27	CTAP2_ERR_OPERATION_DENIED	Not authorized for requested operation.
0x28	CTAP2_ERR_KEY_STORE_FULL	Internal key storage is full.
0x2B	CTAP2_ERR_UNSUPPORTED_OPTION	Unsupported option.
0x2C	CTAP2_ERR_INVALID_OPTION	Not a valid option for current operation.
0x2D	CTAP2_ERR_KEEPALIVE_CANCEL	Pending keep alive was cancelled.
0x2E	CTAP2_ERR_NO_CREDENTIALS	No valid credentials provided.
0x2F	CTAP2_ERR_USER_ACTION_TIMEOUT	A user action timeout occurred.
0x30	CTAP2_ERR_NOT_ALLOWED	Continuation command, such as, authenticatorGetNextAssertion not allowed.
0x31	CTAP2_ERR_PIN_INVALID	PIN Invalid.
0x32	CTAP2_ERR_PIN_BLOCKED	PIN Blocked.
0x33	CTAP2_ERR_PIN_AUTH_INVALID	PIN authentication,pinUvAuthParam, verification failed.
0x34	CTAP2_ERR_PIN_AUTH_BLOCKED	PIN authentication using pinUvAuthToken blocked. Requires power cycle to reset.
0x35	CTAP2_ERR_PIN_NOT_SET	No PIN has been set.
0x36	CTAP2_ERR_PUAT_REQUIRED	A pinUvAuthToken is required for the selected operation. See also the pinUvAuthToken option ID.
0x37	CTAP2_ERR_PIN_POLICY_VIOLATION	PIN policy violation. Minimum PIN length or PIN complexity may trigger this error. The platform should check the minimum PIN length in authenticatorGetInfo to discriminate between the causes of this error.
0x38	*Reserved for Future Use*	*Reserved for Future Use*
0x39	CTAP2_ERR_REQUEST_TOO_LARGE	Authenticator cannot handle this request due to memory constraints.
0x3A	CTAP2_ERR_ACTION_TIMEOUT	The current operation has timed out.
0x3B	CTAP2_ERR_UP_REQUIRED	User presence is required for the requested operation.
0x3C	CTAP2_ERR_UV_BLOCKED	built-in user verification is disabled.
0x3D	CTAP2_ERR_INTEGRITY_FAILURE	A checksum did not match.
0x3E	CTAP2_ERR_INVALID_SUBCOMMAND	The requested subcommand is either invalid or not implemented.
0x3F	CTAP2_ERR_UV_INVALID	built-in user verification unsuccessful. The platform SHOULD retry.
0x40	CTAP2_ERR_UNAUTHORIZED_PERMISSION	The permissions parameter contains an unauthorized permission.
0x7F	CTAP1_ERR_OTHER	Other unspecified error.
0xDF	CTAP2_ERR_SPEC_LAST	CTAP 2 spec last error.
0xE0	CTAP2_ERR_EXTENSION_FIRST	Extension specific error.
0xEF	CTAP2_ERR_EXTENSION_LAST	Extension specific error.
0xF0	CTAP2_ERR_VENDOR_FIRST	Vendor specific error.
0xFF	CTAP2_ERR_VENDOR_LAST	Vendor specific error.

8.3. Utility functions§

This protocol uses the following utility functions for encoding various values in various algorithms:

uint8(x)
Returns the least-significant eight bits of x as a single byte.

uint32LittleEndian(x)
Returns a sequence of four bytes whose values are the least-significant eight bits of x, x >> 8, x >> 16, and x >> 24, respectively.

uint64LittleEndian(x)
Returns a sequence of eight bytes whose values are the least-significant eight bits of x, x >> 8, x >> 16, x >> 24, x >> 32, x >> 40, x >> 48, x >> 56, respectively.

9. Mandatory features§

Authenticators that include `FIDO_2_2` in versions:

1. MUST support the hmac-secret extension.
2. MUST support PIN establishment/maintenance or a built-in user verification method (or both) if the option ID for rk has the value `true`. The option ID values for clientPin and uv MUST have either the values `true` or `false`, depending on if a pin has been set or a biometric template enrolled on the authenticator.
3. MUST either include the credMgmt option ID with the value `true` in the authenticatorGetInfo response's options member, or support all the same functionality via a built-in UI, if the rk option ID has the value `true`.
4. MUST support the credProtect extension if some form of user verification is supported, unless all credentials are implicitly created at credProtect level three.
5. MUST include the pinUvAuthToken option ID with the value `true` in the authenticatorGetInfo response's options member if either the clientPin or uv option IDs have the value `true`.
6. MUST include an array element with the value 2 in the authenticatorGetInfo response's pinUvAuthProtocols member (i.e. support PIN/UV auth protocol two) if it includes any values at all.

10. Interoperating with CTAP1/U2F authenticators§

This section defines:

1. How a platform maps a subset of CTAP2 requests to CTAP1/U2F requests and, conversely, how it maps the CTAP1/U2F responses to CTAP2 responses. (Only requests that do not require CTAP2-only features can be so mapped.)
2. How RPs verify CTAP1/U2F-based authenticatorMakeCredential and authenticatorGetAssertion responses.
3. How authenticators allow credentials to be exposed via both CTAP2 and CTAP1/U2F.

Platforms MAY implement support for CTAP1/U2F, but authenticators SHOULD support it. Not supporting U2F may result in an authenticator that does not function on all websites and thus may appear to be broken to users. Thus authenticators that do not support CTAP1/U2F are not suitable for sale to the general public but may be manufactured for specific cases where it is known that CTAP1/U2F support is unnecessary.

10.1. Framing of U2F commands§

The U2F protocol is based on a request-response mechanism, where a requester sends a request message to a U2F device, which always results in a response message being sent back from the U2F device to the requester.

The request message has to be "framed" to send to the lower layer. Taking the signature request as an example, the "framing" is a way for the FIDO client to tell the lower transport layer that it is sending a signature request and then send the raw message contents. The framing also specifies how the transport will carry back the response raw message and any meta-information such as an error code if the command failed.

In this current version of U2F, the framing is defined based on the ISO7816-4:2005 extended APDU format. This is very appropriate for the USB transport since devices are typically built around secure elements which understand this format already. This same argument may apply for futures such as Bluetooth based devices. For other futures based on other transports, such as a built-in u2f token on a mobile device TEE, this framing may not be appropriate, and a different framing may need to be defined.

10.1.1. U2F Request Message Framing§

The raw request message is framed as a command APDU:

CLA INS P1 P2 LC1 LC2 LC3

Where:

CLA: Reserved to be used by the underlying transport protocol (if applicable). The host application SHALL set this byte to zero.

INS: U2F command code, defined in the following sections.

P1, P2: Parameter 1 and 2, defined by each command.

LC1-LC3: Length of the request data, big-endian coded, i.e. LC1 being MSB and LC3 LSB

10.1.2. U2F Response Message Framing

The raw response data is framed as a response APDU:

SW1 SW2

Where:

SW1, SW2: Status word bytes 1 and 2, forming a 16-bit status word, defined below. SW1 is MSB and SW2 LSB.

Status Codes

The following ISO7816-4 defined status words have a special meaning in U2F:

SW_NO_ERROR: The command completed successfully without error.

SW_CONDITIONS_NOT_SATISFIED: The request was rejected due to test-of-user-presence being required.

SW_WRONG_DATA: The request was rejected due to an invalid key handle.

SW_COMMAND_NOT_ALLOWED: The command is not allowed at this time, e.g. because U2F is disabled.

Each implementation may define any other vendor-specific status codes, providing additional information about an error condition. Only the error codes listed above will be handled by U2F FIDO clients, whereas others will be seen as general errors and logging of these is OPTIONAL.

10.2. Using the CTAP2 authenticatorMakeCredential Command with CTAP1/U2F authenticators

Platform follows the following procedure (Fig: Mapping: WebAuthn authenticatorMakeCredential to and from CTAP1/U2F Registration Messages):

1. Platform tries to get information about the authenticator by sending authenticatorGetInfo command as specified in CTAP2 protocol overview.
    - CTAP1/U2F authenticator returns a command error or improperly formatted CBOR response. For any failure, platform MAY fall back to CTAP1/U2F protocol.
2. Map CTAP2 authenticatorMakeCredential request to U2F_REGISTER request.
    - Platform verifies that CTAP2 request does not have any parameters that CTAP1/U2F authenticators cannot fulfill.
        - All of the below conditions MUST be true for the platform to proceed to next step. If any of the below conditions is not true, platform errors out with CTAP2_ERR_UNSUPPORTED_OPTION.
            - pubKeyCredParams MUST use the ES256 algorithm (-7).
            - Options MUST NOT include "rk" set to true.
            - Options MUST NOT include "uv" set to true.
        - If excludeList is not empty:
            - If the excludeList is not empty, the platform MUST send signing request with check-only control byte to the CTAP1/U2F authenticator using each of the credential ids (key handles) in the excludeList. If any of them does not result in an error, that means that this is a known device. Afterwards, the platform MUST still send a dummy registration request (with a dummy appid and invalid challenge) to CTAP1/U2F authenticators that it believes are excluded. This makes it so the user still needs to touch the CTAP1/U2F authenticator before the RP gets told that the token is already registered.
    - Use clientDataHash parameter of CTAP2 request as CTAP1/U2F challenge parameter (32 bytes).
    - Let `rpIdHash` be a byte string of size 32 initialized with SHA-256 hash of `rp.id` parameter as CTAP1/U2F application parameter (32 bytes).
3. Send the U2F_REGISTER request to the authenticator as specified in [U2FRawMsgs] spec.
4. If the authenticator response message contains the status code SW_COMMAND_NOT_ALLOWED, U2F is disabled at this time. Abandon this operation. The platform SHOULD retry using CTAP2 if present in the versions array.
5. Map the U2F registration response message (see: FIDO U2F Raw Message Formats v1.2 § registration-response-message-success) to a CTAP2 authenticatorMakeCredential response message:

- Generate `authenticatorData` from the U2F registration response message (FIDO U2F Raw Message Formats v1.2 § registration-response-message-success) received from the authenticator:
  - Initialize `attestedCredData`:
    - Let `credentialIdLength` be a 2-byte unsigned big-endian integer representing length of the Credential ID initialized with CTAP1/U2F response key handle length.
    - Let `credentialId` be a `credentialIdLength` byte string initialized with CTAP1/U2F response key handle bytes.
    - Let `x9encodedUserPublicKey`be the `user public key` returned in the U2F registration response message [U2FRawMsgs]. Let `coseEncodedCredentialPublicKey` be the result of converting `x9encodedUserPublicKey`'s value from ANS X9.62 / Sec-1 v2 uncompressed curve point representation [SEC1V2] to COSE_Key representation ([RFC9052] Section 7).
    - Let `attestedCredData` be a byte string with following structure:

Length (in bytes)	Description	Value
16	The AAGUID of the authenticator.	Initialized with all zeros.
2	Byte length L of Credential ID	Initialized with `credentialIdLength` bytes.
`credentialIdLength`	Credential ID.	Initialized with `credentialId` bytes.
77	The credential public key.	Initialized with `coseEncodedCredentialPublicKey` bytes.

  - Initialize `authenticatorData`:
    - Let `flags` be a byte whose zeroth bit (bit 0, UP) is set, and whose sixth bit (bit 6, AT) is set, and all other bits are zero (bit zero is the least significant bit). See also Authenticator Data section of [WebAuthn].
    - Let `signCount` be a 4-byte unsigned integer initialized to zero.
    - Let `authenticatorData` be a byte string with the following structure:

Length (in bytes)	Description	Value
32	SHA-256 hash of the `rp.id`.	Initialized with `rpIdHash` bytes.
1	Flags	Initialized with `flags`' value.
4	Signature counter (signCount).	Initialized with `signCount` bytes.
Variable Length	Attested credential data.	Initialized with `attestedCredData`'s value.

- Let `attestationStatement` be a CBOR map (see "attStmtTemplate" in Generating an Attestation Object [WebAuthn]) with the following keys, whose values are as follows:
  - Set "x5c" as an array of the one attestation cert extracted from CTAP1/U2F response.
  - Set "sig" to be the "signature" bytes from the U2F registration response message [U2FRawMsgs]. Note: An ASN.1-encoded ECDSA signature value ranges over 8–72 bytes in length. [U2FRawMsgs] incorrectly states a different length range.
- Let `attestationObject` be a CBOR map (see "attObj" in Generating an Attestation Object [WebAuthn]) with the following keys, whose values are as follows:
  - Set "authData" to `authenticatorData`.
  - Set "fmt" to "fido-u2f".
  - Set "attStmt" to `attestationStatement`.

6. Return `attestationObject` to the caller.

EXAMPLE 4

Sample CTAP2 authenticatorMakeCredential Request (CBOR):

```
{1: h'687134968222EC17202E42505F8ED2B16AE22F16BB05B88C25DB9E602645F141',
 2: {"id": "example.com",
     "name": "example.com"},
 3: {"id": "1098237235409872",
     "name": "johnpsmith@example.com",
     "icon": "https://pics.example.com/00/p/aBjjjpqPb.png",
     "displayName": "John P. Smith"}
```

```
     displayName": "John P. Smith"},
 4: [{"type": "public-key", "alg": -7},
     {"type": "public-key", "alg": -257}]}
```

CTAP1/U2F Request from above CTAP2 authenticatorMakeCredential request

```
687134968222EC17202E42505F8ED2B16AE22F16BB05B88C25DB9E602645F141    # clientDataHash
A379A6F6EEAFB9A55E378C118034E2751E682FAB9F2D30AB13D2125586CE1947    # rpIdHash
```

Sample CTAP1/U2F Response from the device

```
05                                                                  # Reserved Byte (1 Byte
)
04E87625896EE4E46DC032766E8087962F36DF9DFE8B567F3763015B1990A60E    # User Public Key (65 B
tes)
1427DE612D66418BDA1950581EBC5C8C1DAD710CB14C22F8C97045F4612FB20C    # ...
91                                                                  # ...
40                                                                  # Key Handle Length (1
yte)
3EBD89BF77EC509755EE9C2635EFAAAC7B2B9C5CEF1736C3717DA48534C8C6B6    # Key Handle (Key Handl
 Length Bytes)
54D7FF945F50B5CC4E78055BDD396B64F78DA2C5F96200CCD415CD08FE420038    # ...
3082024A30820132A0030201020204046C8822300D06092A864886F70D01010B    # X.509 Cert (Variable
ength Cert)
0500302E312C302A06035504031323597567269636F2055324620526F6F742043    # ...
412053657269616C2034353732303036333130201 70D31343038303130303030    # ...
30305A180F32303530303930343030303030305A302C312A302806035504030C    # ...
2159756269636F2055324620454520536572696 16C2032343931383233323437    # ...
37303059301306072A8648CE3D020106082A8648CE3D030107034200043CCAB9    # ...
2CCB97287EE8E639437E21FCD6B6F165B2D5A3F3DB131D31C16B742BB476D8D1    # ...
E99080EB546C9BBDF556E6210FD42785899E78CC589EBE310F6CDB9FF4A33B30    # ...
39302206092B0601040182C40A020415312E332E362E312E342E312E34313438    # ...
322E312E323013060B2B0601040182E51C020101040403020430300D06092A86    # ...
4886F70D01010B050003820101009F9B052248BC4CF42CC5991FCAABAC9B651B    # ...
BE5BDCDC8EF0AD2C1C1FFB36D18715D42E78B249224F92C7E6E7A05C49F0E7E4    # ...
C881BF2E94F45E4A21833D7456851D0F6C145A29540C874F3092C934B43D222B    # ...
8962C0F410CEF1DB75892AF116B44A96F5D35ADEA3822FC7146F6004385BCB69    # ...
B65C99E7EB6919786703C0D8CD41E8F75CCA44AA8AB725AD8E799FF3A8696A6F    # ...
1B2656E631B1E40183C08FDA53FA4A8F85A05693944AE179A1339D002D15CABD    # ...
810090EC722EF5DEF9965A371D415D624B68A2707CAD97BCDD1785AF97E258F3    # ...
3DF56A031AA0356D8E8D5EBCADC74E071636C6B110ACE5CC9B90DFEACAE640FF    # ...
1BB0F1FE5DB4EFF7A95F060733F5                                        # ...
30450220324779C68F3380288A1197B6095F7A6EB9B1B1C127F66AE12A99FE85    # Signature (variable l
ngth)
32EC23B9022100E39516AC4D61EE64044D50B415A6A4D4D84BA6D895CB5AB7A1    # ...
AA7D081DE341FA                                                      # ...
```

Authenticator Data from CTAP1/U2F Response

```
A379A6F6EEAFB9A55E378C118034E2751E682FAB9F2D30AB13D2125586CE1947    # rpIdHash
41                                                                  # flags
00000000                                                            # Sign Count
00000000000000000000000000000000                                    # AAGUID
0040                                                                # Key Handle Length (1
yte)
3EBD89BF77EC509755EE9C2635EFAAAC7B2B9C5CEF1736C3717DA48534C8C6B6    # Key Handle (Key Handl
 Length Bytes)
54D7FF945F50B5CC4E78055BDD396B64F78DA2C5F96200CCD415CD08FE420038    # ...
A501020326200121582 0E87625896EE4E46DC032766E8087962F36DF9DFE8B56    # Public Key
7F3763015B1990A60E1422582027DE612D66418BDA1950581EBC5C8C1DAD710C    # ...
B14C22F8C97045F4612FB20C91                                          # ...
```

Mapped CTAP2 authenticatorMakeCredential response(CBOR)

```
{1: "fido-u2f",
 2: h'A379A6F6EEAFB9A55E378C118034E2751E682FAB9F2D30AB13D2125586CE1947
      410000000000000000000000000000000000000000000000403EBD89BF77EC509755
      EE9C2635EFAAAC7B2B9C5CEF1736C3717DA48534C8C6B654D7FF945F50B5CC4E
      78055BDD396B64F78DA2C5F96200CCD415CD08FE420038A50102032620012158
      20E87625896EE4E46DC032766E8087962F36DF9DFE8B567F3763015B1990A60E
      1422582027DE612D66418BDA1950581EBC5C8C1DAD710CB14C22F8C97045F461
      2FB20C91',
 3: {"sig": h'30450220324779C68F3380288A1197B6095F7A6EB9B1B1C127F66AE12A99FE85
             32EC23B9022100E39516AC4D61EE64044D50B415A6A4D4D84BA6D895CB5AB7A1
             AA7D081DE341FA',
            "x5c": [h'3082024A30820132A0030201020204046C8822300D06092A864886F70D01010B
                      0500302E312C302A06035504031323597567269636F2055324620526F6F742043
                      412053657269616C2034353732303036333130201 70D31343038303130303030
                      30305A180F32303530303930343030303030305A302C312A3028060355040 30C
```

```
2159756269636F20553246204545205365726961 6C20323439313832333234373
7303059301306072A8648CE3D020106082A8648CE3D030107034200043CCAB92CCB97287EE8E639437E21FCD6B6F165B2D5A3F3DB131D31C16B742BB476D8D1E99080EB546C9BBDF556E6210FD42785899E78CC589EBE310F6CDB9FF4A33B3039302206092B0601040182C40A020415312E332E362E312E342E312E34313438322E312E323013060B2B0601040182E51C020101040403020430300D06092A864886F70D01010B050003820101009F9B052248BC4CF42CC5991FCAABAC9B651BBE5BDCDC8EF0AD2C1C1FFB36D18715D42E78B249224F92C7E6E7A05C49F0E7E4C881BF2E94F45E4A21833D7456851D0F6C145A29540C874F3092C934B43D222B8962C0F410CEF1DB75892AF116B44A96F5D35ADEA3822FC7146F6004385BCB69B65C99E7EB6919786703C0D8CD41E8F75CCA44AA8AB725AD8E799FF3A8696A6F1B2656E631B1E40183C08FDA53FA4A8F85A05693944AE179A1339D002D15CABD810090EC722EF5DEF9965A371D415D624B68A2707CAD97BCDD1785AF97E258F33DF56A031AA0356D8E8D5EBCADC74E071636C6B110ACE5CC9B90DFEACAE640FF1BB0F1FE5DB4EFF7A95F060733F5']}}
```

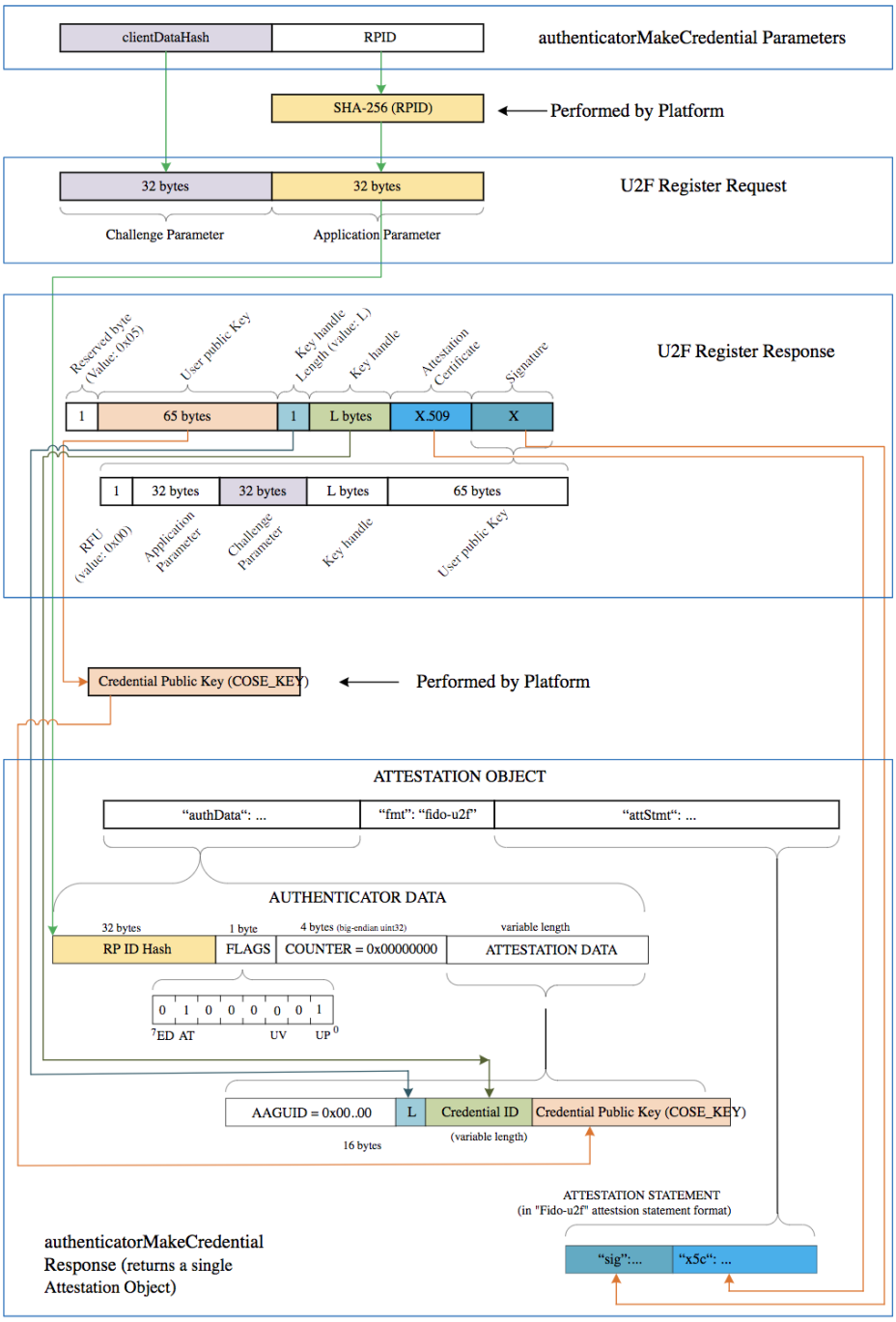

Figure 3 *Mapping: WebAuthn authenticatorMakeCredential to and from CTAP1/U2F Registration Messages.*

10.3. Using the CTAP2 authenticatorGetAssertion Command with CTAP1/U2F authenticators§

Platform follows the following procedure (Fig: Mapping: WebAuthn authenticatorGetAssertion to and from CTAP1/U2F Authentication Messages):

1. Platform tries to get information about the authenticator by sending authenticatorGetInfo command as specified in CTAP2 protocol overview.
    - CTAP1/U2F authenticator returns a command error or improperly formatted CBOR response. For any failure, platform MAY fall back to CTAP1/U2F protocol.
2. Map CTAP2 authenticatorGetAssertion request to U2F_AUTHENTICATE request:
    - Platform verifies that CTAP2 request does not have any parameters that CTAP1/U2F authenticators cannot fulfill:
        - All of the below conditions MUST be true for the platform to proceed to next step. If any of the below conditions is not true, platform errors out with CTAP2_ERR_UNSUPPORTED_OPTION.
            - Options MUST NOT include "uv" set to true.
            - allowList MUST have at least one credential.
    - If allowList has more than one credential, platform has to loop over the list and send individual different U2F_AUTHENTICATE commands to the authenticator. For each credential in credential list, map CTAP2 authenticatorGetAssertion request to U2F_AUTHENTICATE as below:
        - Let `controlByte` be a byte initialized as follows:
            - If "up" is set to false, set it to 0x08 (dont-enforce-user-presence-and-sign).
            - For USB, set it to 0x07 (check-only). This should prevent call getting blocked on waiting for user input. If response returns success, then call again setting the enforce-user-presence-and-sign.
            - For NFC, set it to 0x03 (enforce-user-presence-and-sign). The tap has already provided the presence and won't block.
        - Use clientDataHash parameter of CTAP2 request as CTAP1/U2F challenge parameter (32 bytes).
        - Let `rpIdHash` be a byte string of size 32 initialized with SHA-256 hash of `rp.id` parameter as CTAP1/U2F application parameter (32 bytes).
        - Let `credentialId` is the byte string initialized with the id for this PublicKeyCredentialDescriptor.
        - Let `keyHandleLength` be a byte initialized with length of `credentialId` byte string.
        - Let `u2fAuthenticateRequest` be a byte string with the following structure:

Length (in bytes)	Description	Value
32	Challenge parameter	Initialized with clientDataHash parameter bytes.
32	Application parameter	Initialized with `rpIdHash` bytes.
1	Key handle length	Initialized with `keyHandleLength`'s value.
`keyHandleLength`	Key handle	Initialized with `credentialId` bytes.

        and let `Control Byte` be P1 of the framing.
3. Send `u2fAuthenticateRequest` to the authenticator.
4. If the authenticator response message contains the status code SW_COMMAND_NOT_ALLOWED, U2F is disabled at this time. Abandon this operation. The platform SHOULD retry using CTAP2.
5. Map the U2F authentication response message (see the "Authentication Response Message: Success" section of [U2FRawMsgs]) to a CTAP2 authenticatorGetAssertion response message:
    - Generate `authenticatorData` from the U2F authentication response message received from the authenticator:
        - Copy bits 0 (the UP bit) and bit 1 from the CTAP2/U2F response user presence byte to bits 0 and 1 of the CTAP2 flags, respectively. Set all other bits of flags to zero. Note: bit zero is the least significant bit. See also Authenticator Data section of [WebAuthn].
        - Let `signCount` be a 4-byte unsigned integer initialized with CTAP1/U2F response counter field.
        - Let `authenticatorData` is a byte string of following structure:

Length (in bytes)	Description	Value
32	SHA-256 hash of the `rp.id`.	Initialized with `rpIdHash` bytes.
1	Flags	Initialized with `flags`' value.
4	Signature counter (signCount)	Initialized with `signCount` bytes.

    - Let `authenticatorGetAssertionResponse` be a CBOR map with the following keys whose values are as follows:

- Set 0x01 with the credential from allowList that whose response succeeded.
- Set 0x02 with `authenticatorData` bytes.
- Set 0x03 with signature field from CTAP1/U2F authentication response message. Note: An ASN.1-encoded ECDSA signature value ranges over 8–72 bytes in length. [U2FRawMsgs] incorrectly states a different length range.

EXAMPLE 5

Sample CTAP2 authenticatorGetAssertion Request (CBOR):

```
{1: "example.com",
 2: h'687134968222EC17202E42505F8ED2B16AE22F16BB05B88C25DB9E602645F141',
 3: [{"type": "public-key",
      "id": h'3EBD89BF77EC509755EE9C2635EFAAAC7B2B9C5CEF1736C3717DA48534C8C6B6
              54D7FF945F50B5CC4E78055BDD396B64F78DA2C5F96200CCD415CD08FE420038'}],
 5: {"up": true}}
```

CTAP1/U2F Request from above CTAP2 authenticatorGetAssertion request

```
687134968222EC17202E42505F8ED2B16AE22F16BB05B88C25DB9E602645F141    # clientDataHash
A379A6F6EEAFB9A55E378C118034E2751E682FAB9F2D30AB13D2125586CE1947    # rpIdHash
40                                                                  # Key Handle Length (1
yte)
3EBD89BF77EC509755EE9C2635EFAAAC7B2B9C5CEF1736C3717DA48534C8C6B6    # Key Handle (Key Handl
 Length Bytes)
54D7FF945F50B5CC4E78055BDD396B64F78DA2C5F96200CCD415CD08FE420038    # ...
```

Sample CTAP1/U2F Response from the device

```
01                                                                  # User Presence (1 Byte
)
0000003B                                                            # Sign Count (4 Bytes
)
304402207BDE0A52AC1F4C8B27E003A370CD66A4C7118DD22D5447835F45B99C    # Signature (variable l
ngth)
68423FF702203C517B47877F85782DE10086A783D1E7DF4E3639E771F5F6AFA3    # ...
5AAD5373858E                                                        # ...
```

Authenticator Data from CTAP1/U2F Response

```
A379A6F6EEAFB9A55E378C118034E2751E682FAB9F2D30AB13D2125586CE1947    # rpIdHash
01                                                                  # User Presence (1 Byte
)
0000003B                                                            # Sign Count (4 Bytes
)
```

Mapped CTAP2 authenticatorGetAssertion response(CBOR)

```
{1: {"type": "public-key",
     "id": h'3EBD89BF77EC509755EE9C2635EFAAAC7B2B9C5CEF1736C3717DA48534C8C6B6
             54D7FF945F50B5CC4E78055BDD396B64F78DA2C5F96200CCD415CD08FE420038'},
 2: h'A379A6F6EEAFB9A55E378C118034E2751E682FAB9F2D30AB13D2125586CE1947
      010000003B',
 3: h'304402207BDE0A52AC1F4C8B27E003A370CD66A4C7118DD22D5447835F45B99C
      68423FF702203C517B47877F85782DE10086A783D1E7DF4E3639E771F5F6AFA3
      5AAD5373858E'}
```

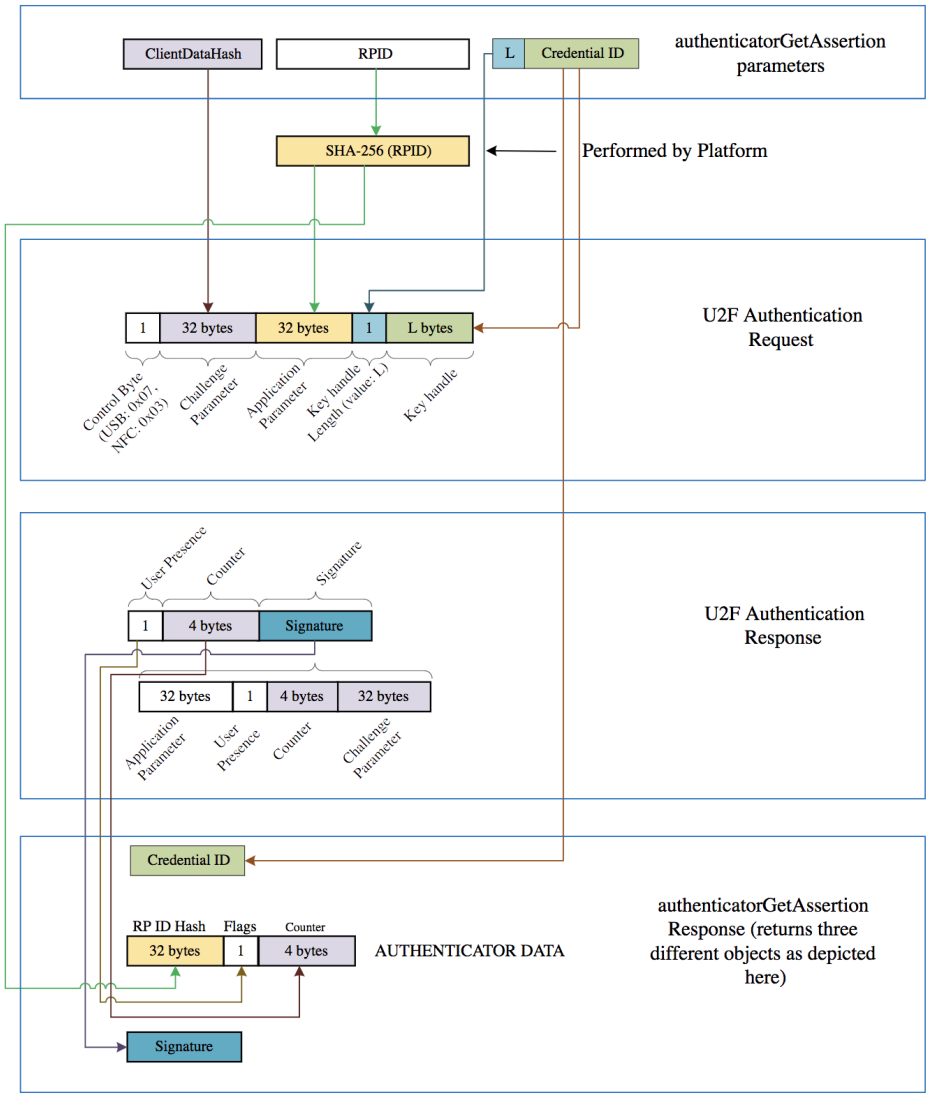

Figure 4 *Mapping: WebAuthn authenticatorGetAssertion to and from CTAP1/U2F Authentication Messages.*

10.4. Cross-version Credential Compatibility§

If an authenticator supports both CTAP1/U2F and CTAP2 then a credential created using CTAP1/U2F MUST be assertable over CTAP2. (Credentials created over CTAP1/U2F MUST NOT be discoverable credentials though.) From § 10.3 Using the CTAP2 authenticatorGetAssertion Command with CTAP1/U2F authenticators this means that an authenticator MUST accept, over CTAP2, the credential ID of a credential that was created using U2F where the application parameter at the time of creation was the SHA-256 digest of the RP ID that is given at assertion time.

11. Transport-specific Bindings§

11.1. Secure protocol implementation§

In order to ensure that the interaction between the platform and any authenticators is secure, authenticators SHALL:

- Ensure that all state (e.g. discoverable credentials, signature counters, PINs, etc) that is observable or alterable over FIDO interfaces is not observable or alterable over any other interfaces on transports that FIDO has defined.
- Ensure that all non-discoverable credentials that are created over FIDO interfaces are not valid over any other interfaces on transports that FIDO has defined. (For example, if non-discoverable credentials store state in the credential ID, protected by an authenticator-global secret, then that secret MUST only be used for requests received over FIDO interfaces.)

> NOTE: Above recommendations are also valid for future transports.

FIDO interfaces are defined as:

- USB, when using USB HID and the FIDO_USAGE_PAGE/FIDO_USAGE_CTAPHID combination.

- LIGHTNING, when using USB HID and the FIDO_USAGE_PAGE/FIDO_USAGE_CTAPHID combination, tunnelled over the Apple Interface Accessory Protocol.
- NFC, when the applet is selected as specified.
  - Authenticator SHALL NOT allow FIDO applet to be implicitly selected or enabled.
    - Recommended: Authenticator SHALL NOT have default applet selected on power cycle. All CTAP commands SHALL be preceded by an explicit applet selection command as described in Applet selection section.
    - Alternative: If authenticator has a FIDO applet selected for some reason at power cycle, it SHALL be in disabled mode and SHALL ONLY be enabled once it receives explicit applet selection command as described in Applet selection section.
  - Authenticator SHALL disable FIDO interface when it receives applet deselect command.
- BLE, when using the FIDO GATT service.
- HYBRID, when using the FIDO Hybrid service.

11.2. USB Human Interface Device (USB HID)

See also § 11.1 Secure protocol implementation.

11.2.1. Design rationale

CTAP messages are framed for USB transport using the HID (Human Interface Device) protocol. We henceforth refer to the protocol as CTAPHID. The CTAPHID protocol is designed with the following design objectives in mind

- Driver-less installation on all major host platforms
- Multi-application support with concurrent application access without the need for serialization and centralized dispatching.
- Fixed latency response and low protocol overhead
- Scalable method for CTAPHID device discovery

Since HID data is sent as interrupt packets and multiple applications may access the HID stack at once, a non-trivial level of complexity has to be added to handle this.

11.2.2. Protocol structure and data framing

The CTAP protocol is designed to be concurrent and state-less in such a way that each performed function is not dependent on previous actions. However, there has to be some form of "atomicity" that varies between the characteristics of the underlying transport protocol, which for the CTAPHID protocol introduces the following terminology:

- Transaction
- Message
- Packet

A **transaction** is the highest level of aggregated functionality, which in turn consists of a request, followed by a response message. Once a request has been initiated, the transaction has to be entirely completed or aborted before a second transaction can take place and a response is never sent without a previous request. Transactions exist only at the highest CTAP protocol layer.

Request and response **messages** are in turn divided into individual fragments, known as **packets**. The packet is the smallest form of protocol data unit, which in the case of CTAPHID are mapped into HID reports.

11.2.3. Concurrency and channels

Additional logic and overhead is required to allow a CTAPHID device to deal with multiple "clients", i.e. multiple applications accessing the single resource through the HID stack. Each client communicates with a CTAPHID device through a logical **channel**, where each application uses a unique 32-bit **channel identifier** for routing and arbitration purposes.

A channel identifier is allocated by the FIDO authenticator to ensure its system-wide uniqueness. The actual algorithm for generation of channel identifiers is vendor specific and not defined by this specification.

Channel ID 0 is reserved and `0xffffffff` is reserved for broadcast commands, i.e. at the time of channel allocation.

11.2.4. Message and packet structure

Packets are one of two types, **initialization packets** and **continuation packets**. As the name suggests, the first packet sent in a message is an initialization packet, which also becomes the start of a transaction. If the entire message does not fit into one packet (including the CTAPHID protocol overhead), one or more continuation packets have to be sent in strict ascending order to complete the message transfer.

A message sent from a host to a device is known as a**request** and a message sent from a device back to the host is known as a **response**. A request always triggers a response and response messages are never sent ad-hoc, i.e. without a prior request message. However, a keep-alive message can be sent between a request and a response message.

The request and response messages have an identical structure. A transaction is started with the initialization packet of the request message and ends with the last packet of the response message. The client starting a transaction may also abort it.

Packets are always fixed size (defined by the endpoint and HID report descriptors) and although all bytes may not be needed in a particular packet, the full size always has to be sent. Unused bytes SHOULD be set to zero.

An initialization packet is defined as

Offset	Length	Mnemonic	Description
0	4	CID	Channel identifier
4	1	CMD	Command identifier (bit 7 always set)
5	1	BCNTH	High part of payload length
6	1	BCNTL	Low part of payload length
7	(s - 7)	DATA	Payload data (s is equal to the fixed packet size)

The command byte has always the highest bit set to distinguish it from a continuation packet, which is described below.

A continuation packet is defined as

Offset	Length	Mnemonic	Description
0	4	CID	Channel identifier
4	1	SEQ	Packet sequence 0x00..0x7f (bit 7 always cleared)
5	(s - 5)	DATA	Payload data (s is equal to the fixed packet size)

With this approach, a message with a payload less or equal to (s - 7) may be sent as one packet. A larger message is then divided into one or more continuation packets, starting with sequence number 0, which then increments by one to a maximum of 127.

With a packet size of 64 bytes (max for full-speed devices), this means that the maximum message payload length is 64 - 7 + 128 * (64 - 5) = 7609 bytes.

11.2.5. Arbitration

In order to handle multiple channels and clients concurrency, the CTAPHID protocol has to maintain certain internal states, block conflicting requests and maintain protocol integrity. The protocol relies on each client application (channel) behaves politely, i.e. does not actively act to destroy for other channels. With this said, a malign or malfunctioning application can cause issues for other channels. Expected errors and potentially stalling applications should however, be handled properly.

11.2.5.1. Transaction atomicity, idle and busy states.

A transaction always consists of three stages:

1. A message is sent from the host to the device
2. The device processes the message
3. A response is sent back from the device to the host

The protocol is built on the assumption that a plurality of concurrent applications may try ad-hoc to perform transactions at any time, with each transaction being atomic, i.e. it cannot be interrupted by another application once started.

The application channel that manages to get through the first initialization packet when the device is in idle state will keep the device locked for other channels until the last packet of the response message has been received or the transaction is aborted. The device then returns to idle state, ready to perform another transaction for the same or a different channel. Between two transactions, the device might need to keep some state. A host

application MUST assume that any other process may execute other transactions at any time and former state will be dropped.

If an application tries to access the device from a different channel while the device is busy with a transaction, that request will immediately fail with a busy-error message sent to the requesting channel.

11.2.5.2. Transaction timeout

A transaction has to be completed within a specified period of time to prevent a stalling application to cause the device to be completely locked out for access by other applications. If for example an application sends an initialization packet that signals that continuation packets will follow and that application crashes, the device will back out that pending channel request and return to an idle state.

11.2.5.3. Transaction abort and re-synchronization

If an application for any reason "gets lost", gets an unexpected response or error, it MAY at any time issue an abort-and-resynchronize command. If the device detects an INIT command during a transaction that has the same channel id as the active transaction, the transaction is aborted (if possible) and all buffered data flushed (if any). The device then returns to idle state to become ready for a new transaction.

If an application wishes to abort a command after the request has been fully sent, e.g. while an authenticator is waiting for user presence, the application MAY do this by sending a CTAPHID_CANCEL command.

11.2.5.4. Packet sequencing

The device keeps track of packets arriving in correct and ascending order and that no expected packets are missing. The device will continue to assemble a message until all parts of it has been received or that the transaction times out. Spurious continuation packets appearing without a prior initialization packet will be ignored.

11.2.6. Channel locking

In order to deal with aggregated transactions that may not be interrupted, such as tunneling of vendor-specific commands, a channel lock command MAY be implemented. By sending a channel lock command, the device prevents other channels from communicating with the device until the channel lock has timed out or been explicitly unlocked by the application.

This feature is optional and has not to be considered by general CTAP HID applications.

11.2.7. Protocol version and compatibility

The CTAPHID protocol is designed to be extensible yet maintain backwards compatibility, to the extent it is applicable. This means that a CTAPHID host SHALL support any version of a device with the command set available in that particular version.

11.2.8. HID device implementation

This description assumes knowledge of the USB and HID specifications and is intended to provide the basics for implementing a CTAPHID device. There are several ways to implement USB devices and reviewing these different methods is beyond the scope of this document. This specification targets the interface part, where a device is regarded as either a single or multiple interface (composite) device.

The description further assumes (but is not limited to) a full-speed USB device (12 Mbit/s). Although not excluded per se, USB low-speed devices are not practical to use given the 8-byte report size limitation together with the protocol overhead.

11.2.8.1. Interface and endpoint descriptors

The device implements two endpoints (except the control endpoint 0), one for IN and one for OUT transfers. The packet size is vendor defined, but the reference implementation assumes a full-speed device with two 64-byte endpoints.

Interface Descriptor

Mnemonic	Value	Description
bNumEndpoints	2	One IN and one OUT endpoint
bInterfaceClass	0x03	HID

Mnemonic	Value	Description
bInterfaceSubClass	0x00	No interface subclass
bInterfaceProtocol	0x00	No interface protocol

Endpoint 1 descriptor

Mnemonic	Value	Description
bmAttributes	0x03	Interrupt transfer
bEndpointAdress	0x01	1, OUT
bMaxPacketSize	64	64-byte packet max
bInterval	5	Poll every 5 millisecond

Endpoint 2 descriptor

Mnemonic	Value	Description
bmAttributes	0x03	Interrupt transfer
bEndpointAdress	0x81	1, IN
bMaxPacketSize	64	64-byte packet max
bInterval	5	Poll every 5 millisecond

The actual endpoint order, intervals, endpoint numbers and endpoint packet size may be defined freely by the vendor and the host application is responsible for querying these values and handle these accordingly. For the sake of clarity, the values listed above are used in the following examples.

11.2.8.2. HID report descriptor and device discovery

A HID report descriptor is required for all HID devices, even though the reports and their interpretation (scope, range, etc.) makes very little sense from an operating system perspective. The CTAPHID just provides two "raw" reports, which basically map directly to the IN and OUT endpoints. However, the HID report descriptor has an important purpose in CTAPHID, as it is used for device discovery.

For the sake of clarity, a bit of high-level C-style abstraction is provided

```
EXAMPLE 6
// HID report descriptor

const uint8_t HID_ReportDescriptor[] = {
  HID_UsagePage ( FIDO_USAGE_PAGE ),
  HID_Usage ( FIDO_USAGE_CTAPHID ),
  HID_Collection ( HID_Application ),
  HID_Usage ( FIDO_USAGE_DATA_IN ),
  HID_LogicalMin ( 0 ),
  HID_LogicalMaxS ( 0xff ),
  HID_ReportSize ( 8 ),
  HID_ReportCount ( HID_INPUT_REPORT_BYTES ),
  HID_Input ( HID_Data | HID_Absolute | HID_Variable ),
  HID_Usage ( FIDO_USAGE_DATA_OUT ),
  HID_LogicalMin ( 0 ),
  HID_LogicalMaxS ( 0xff ),
  HID_ReportSize ( 8 ),
  HID_ReportCount ( HID_OUTPUT_REPORT_BYTES ),
  HID_Output ( HID_Data | HID_Absolute | HID_Variable ),
HID_EndCollection
};
```

A unique **Usage Page** is defined (0xF1D0) for the FIDO alliance and under this realm, a CTAPHID**Usage** is defined as well (0x01). During CTAPHID device discovery, all HID devices present in the system are examined and devices that match this usage pages and usage are then considered to be CTAPHID devices.

The length values specified by the `HID_INPUT_REPORT_BYTES` and the `HID_OUTPUT_REPORT_BYTES` should typically match the respective endpoint sizes defined in the endpoint descriptors.

11.2.9. CTAPHID commands

The CTAPHID protocol implements the following commands.

11.2.9.1. Mandatory commands

The following list describes the minimum set of commands required by a CTAPHID device. Optional and vendor-specific commands may be implemented as described in respective sections of this document.

11.2.9.1.1. CTAPHID_MSG (0x03)

This command sends an encapsulated CTAP1/U2F message to the device. The semantics of the data message is defined in the U2F Raw Message Format encoding specification.

Request

CMD	CTAPHID_MSG
BCNT	1..(n + 1)
DATA	U2F command byte
DATA + 1	n bytes of data

Response at success

CMD	CTAPHID_MSG
BCNT	1..(n + 1)
DATA	U2F status code
DATA + 1	n bytes of data

11.2.9.1.2. CTAPHID_CBOR (0x10)

This command sends an encapsulated CTAP CBOR encoded message. The semantics of the data message is defined in the CTAP Message encoding specification. Please note that keep-alive messages MAY be sent from the device to the client before the response message is returned.

Request

CMD	CTAPHID_CBOR
BCNT	1..(n + 1)
DATA	CTAP command byte
DATA + 1	n bytes of CBOR encoded data

Response at success

CMD	CTAPHID_CBOR
BCNT	1..(n + 1)
DATA	CTAP status code
DATA + 1	n bytes of CBOR encoded data

11.2.9.1.3. CTAPHID_INIT (0x06)

This command has two functions.

If sent on an allocated CID, it synchronizes a channel, discarding the current transaction, buffers and state as quickly as possible. It will then be ready for a new transaction. The device then responds with the CID of the channel it received the INIT on, using that channel.

If sent on the broadcast CID, it requests the device to allocate a unique 32-bit channel identifier (CID) that can be used by the requesting application during its lifetime. The requesting application generates a nonce that is used to match the response. When the response is received, the application compares the sent nonce with the received one. After a positive match, the application stores the received channel id and uses that for subsequent transactions.

To allocate a new channel, the requesting application SHALL use the broadcast channel CTAPHID_BROADCAST_CID (0xFFFFFFFF). The device then responds with the newly allocated channel in the response, using the broadcast channel.

Request

CMD	CTAPHID_INIT
BCNT	8
DATA	8-byte nonce

Response at success

CMD	CTAPHID_INIT
BCNT	17 (see note below)
DATA	8-byte nonce
DATA+8	4-byte channel ID
DATA+12	CTAPHID protocol version identifier
DATA+13	Major device version number
DATA+14	Minor device version number
DATA+15	Build device version number
DATA+16	Capabilities flags

The protocol version identifies the protocol version implemented by the device. This version of the CTAPHID protocol is 2.

A CTAPHID host SHALL accept a response size that is longer than the anticipated size to allow for future extensions of the protocol, yet maintaining backwards compatibility. Future versions will maintain the response structure of the current version, but additional fields may be added.

The meaning and interpretation of the device version number is vendor defined.

The capability flags value is a bitfield where the following bits values are defined. Unused values are reserved for future use and MUST be set to zero by device vendors.

Name	Value	Description
CAPABILITY_WINK	0x01	If set to 1, authenticator implements CTAPHID_WINK function
CAPABILITY_CBOR	0x04	If set to 1, authenticator implements CTAPHID_CBOR function
CAPABILITY_NMSG	0x08	If set to 1, authenticator DOES NOT implement CTAPHID_MSG function

11.2.9.1.4. CTAPHID_PING (0x01)

Sends a transaction to the device, which immediately echoes the same data back. This command is defined to be a uniform function for debugging, latency and performance measurements.

Request

CMD	CTAPHID_PING
BCNT	0..n
DATA	n bytes

Response at success

CMD	CTAPHID_PING
BCNT	n
DATA	N bytes

11.2.9.1.5. CTAPHID_CANCEL (0x11)

Cancel any outstanding requests on this CID. If there is an outstanding request that can be cancelled, the authenticator MUST cancel it and that cancelled request will reply with the error CTAP2_ERR_KEEPALIVE_CANCEL.

As the CTAPHID_CANCEL command is sent during an ongoing transaction, transaction semantics do not apply. Whether a request was cancelled or not, the authenticator MUST NOT reply to the CTAPHID_CANCEL message

itself. The CTAPHID_CANCEL command MAY be sent by the client during ongoing processing of a CTAPHID_CBOR request. The CTAP2_ERR_KEEPALIVE_CANCEL response MUST be the response to that request, not an error response in the HID transport.

A CTAPHID_CANCEL received while no CTAPHID_CBOR request is being processed, or on a non-active CID SHALL be ignored by the authenticator.

CMD	CTAPHID_CANCEL
BCNT	0

11.2.9.1.6. CTAPHID_ERROR (0x3F)

This command code is used in response messages only.

CMD	CTAPHID_ERROR
BCNT	1
DATA	Error code

The following error codes are defined

ERR_INVALID_CMD	0x01	The command in the request is invalid
ERR_INVALID_PAR	0x02	The parameter(s) in the request is invalid
ERR_INVALID_LEN	0x03	The length field (BCNT) is invalid for the request
ERR_INVALID_SEQ	0x04	The sequence does not match expected value
ERR_MSG_TIMEOUT	0x05	The message has timed out
ERR_CHANNEL_BUSY	0x06	The device is busy for the requesting channel. The client SHOULD retry the request after a short delay. Note that the client MAY abort the transaction if the command is no longer relevant.
ERR_LOCK_REQUIRED	0x0A	Command requires channel lock
ERR_INVALID_CHANNEL	0x0B	CID is not valid.
ERR_OTHER	0x7F	Unspecified error

Note: These values are identical to the BLE transport values.

11.2.9.1.7. CTAPHID_KEEPALIVE (0x3B)

This command code is sent while processing a CTAPHID_MSG. It SHOULD be sent at least every 100ms and whenever the status changes. A KEEPALIVE sent by an authenticator does not constitute a response and does therefore not end an ongoing transaction.

CMD	CTAPHID_KEEPALIVE
BCNT	1
DATA	Status code

The following status codes are defined

STATUS_PROCESSING	1	The authenticator is still processing the current request.
STATUS_UPNEEDED	2	The authenticator is waiting for user presence.

11.2.9.2. Optional commands

The following commands are defined by this specification but are optional and does not have to be implemented.

11.2.9.2.1. CTAPHID_WINK (0x08)

The wink command performs a vendor-defined action that provides some visual or audible identification a particular authenticator. A typical implementation will do a short burst of flashes with a LED or something similar. This is useful when more than one device is attached to a computer and there is confusion which device is

paired with which connection.

Request

CMD	CTAPHID_WINK
BCNT	0
DATA	N/A

Response at success

CMD	CTAPHID_WINK
BCNT	0
DATA	N/A

11.2.9.2.2. CTAPHID_LOCK (0x04)§

The lock command places an exclusive lock for one channel to communicate with the device. As long as the lock is active, any other channel trying to send a message will fail. In order to prevent a stalling or crashing application to lock the device indefinitely, a lock time up to 10 seconds MAY be set. An application requiring a longer lock has to send repeating lock commands to maintain the lock.

Request

CMD	CTAPHID_LOCK
BCNT	1
DATA	Lock time in seconds 0..10. A value of 0 immediately releases the lock

Response at success

CMD	CTAPHID_LOCK
BCNT	0
DATA	N/A

11.2.9.3. Vendor specific commands§

A CTAPHID MAY implement additional vendor specific commands that are not defined in this specification, while being CTAPHID compliant. Such commands, if implemented, MUST use a command in the range between CTAPHID_VENDOR_FIRST (0x40) and CTAPHID_VENDOR_LAST (0x7F).

11.3. ISO7816, ISO14443 and Near Field Communication (NFC)§

See also § 11.1 Secure protocol implementation.

11.3.1. Conformance§

Please refer to [ISO7816-4] for APDU definition.

11.3.2. Protocol§

The general protocol between a FIDO2 client and an authenticator over ISO7816/ISO14443 is as follows:

1. Client sends an applet selection command
2. Authenticator replies with success if the applet is present
3. Client sends a command for an operation
4. Authenticator replies with response data or error
5. Return to 3.

Because of timeouts that may otherwise occur on some platforms, it is RECOMMENDED that the authenticators reply to APDU commands within 800 milliseconds.

11.3.3. Applet selection§

NOTE: See also § 11.1 Secure protocol implementation

A successful Select allows the client to know that the applet is present and active. A client SHALL send a Select to the authenticator before any other command.

The FIDO2 AID consists of the following fields:

Field	Value
`RID`	0xA000000647
`PIX`	0x2F0001

The command to select the FIDO applet is:

CLA	INS	P1	P2	Data In	Le
0x00	0xA4	0x04	0x00	AID	Variable

In response to the applet selection command, the FIDO authenticator replies with its version information string in the successful response.

Clients and authenticators MAY support additional selection mechanisms. Clients MUST fall back to the previously defined selection process if the additional selection mechanisms fail to select the applet. Authenticators MUST at least support the previously defined selection process.

Given legacy support for CTAP1/U2F, the client MUST determine the capabilities of the device at the selection stage.

- If the authenticator implements CTAP1/U2F, the version information SHALL be the string "U2F_V2", or 0x5532465f5632, to maintain backwards-compatibility with CTAP1/U2F-only clients.
- If the authenticator ONLY implements CTAP2, the device SHALL respond with "FIDO_2_0", or 0x4649444f5f325f30.
- If the authenticator implements both CTAP1/U2F and CTAP2, the version information SHALL be the string "U2F_V2", or 0x5532465f5632, to maintain backwards-compatibility with CTAP1/U2F-only clients. CTAP2-aware clients MAY then issue a CTAP authenticatorGetInfo command to determine if the device supports CTAP2 or not.

11.3.4. Applet deselection

NOTE: See also § 11.1 Secure protocol implementation

- Authenticator SHALL deselect or disable FIDO applet upon receiving below `NFCCTAP_CONTROL END CTAP_MSG` command.
  - Authenticators SHALL ignore subsequent FIDO CTAP commands until it receives the next explicit FIDO Applet selection command.
  - `NFCCTAP_CONTROL END CTAP_MSG` command is as follows:

CLA	INS	P1	P2
0x80	0x12 (NFCCTAP_CONTROL)	0x01 (End CTAP_MSG Control Byte)	0x00

11.3.5. Framing

Conceptually, framing defines an encapsulation of FIDO2 commands. This encapsulation is done in an APDU following [ISO7816-4]. Authenticators MUST support short and extended length encoding for this APDU. Fragmentation, if needed, is discussed in the following paragraph.

11.3.5.1. Commands

Commands SHALL have the following format:

CLA	INS	P1	P2	Data In	Le
0x80	0x10	0x00	0x00	CTAP Command Byte || CBOR Encoded Data	Variable

11.3.5.2. Response

Response SHALL have the following format in case of success:

Case	Data	Status word
Success	CTAP Status code || Response data	"9000" - Success
Status update	Status data	"9100" - OK When receiving this, the ISO transport layer will immediately issue an NFCCTAP_GETREPONSE command unless a cancel was issued. The ISO transport layer will provide the status data to the higher layers.
Errors		See [ISO7816-4]

11.3.6. Fragmentation

APDU command may hold up to 255 or 65535 bytes of data using short or extended length encoding respectively. APDU response may hold up to 256 or 65536 bytes of data using short or extended length encoding respectively.

Some requests may not fit into a short APDU command, or the expected response may not fit in a short APDU response. For this reason, FIDO2 client MAY encode APDU command in the following way:

- The request MAY be encoded using *extended length* APDU encoding.
- The request MAY be encoded using *short* APDU encoding. If the request does not fit a short APDU command, the client MUST use ISO 7816-4 APDU chaining.

Short APDU Chaining commands SHALL have the following format:

CLA	INS	P1	P2	Data In
0x90	0x10	0x00	0x00	CTAP Payload

EXAMPLE 7

Sample `authenticatorMakeCredential` request using short APDU encoding and chaining mode:

```
01A80158206871349682 22EC17202E42505F8ED2B16AE22F16BB05B88C25DB9E
602645F14102A2626964697465737 42E63746170646E616D6569746573742E63
74617003A362696458202B6689BB18F4169F069FBCDF50CB6EA3C60A861B9A7B
63946983E0B577B78C70646E616D65717465737463746170406374617 02E636F
6D6B646973706C61794E616D656954657374204374617 00483A263616C672664
747970656A7075626C69632D6B6579A263616C6739010064747970656A707562
6C69632D6B6579A263616C67382464747970656A7075626C69632D6B657906A1
6B686D61632D736563726574F507A162726BF50850FC43AAA411D948CC6C3706
8B8DA1D5080901
```

would be sent to authenticator by platform in two short APDU commands:

- APDU command 1:

```
Platform Request:
90 10 00 00
F0
01A80158206871349682 22EC17202E42505F8ED2B16AE22F16BB05B88C25DB9E
602645F14102A2626964697465737 42E63746170646E616D6569746573742E63
74617003A362696458202B6689BB18F4169F069FBCDF50CB6EA3C60A861B9A7B
63946983E0B577B78C70646E616D65717465737463746170406374617 02E636F
6D6B646973706C61794E616D656954657374204374617 00483A263616C672664
747970656A7075626C69632D6B6579A263616C6739010064747970656A707562
6C69632D6B6579A263616C67382464747970656A7075626C69632D6B657906A1
6B686D61632D736563726574F507A162

Authenticator Response:
9000
```

- APDU command 2:

```
Platform Request:
80 10 00 00
17
726BF50850FC43AAA411D948CC6C37068B8DA1D5080901
00

Authenticator Response:
00
A301667061636B6564025900A20021F5FC0B85CD22E60623BCD7D1CA48948909
```

```
249B4776EB515154E57B66AE12C500000055F8A011F38C0A4D15800617111F9E
DC7D0010F4D57B23DD0CB785680CDAA7F7E44F60A501020326200121582 0DF01
7D0B286795BEA153D166A0A15B4F6B67A3AF4A101E10E8496F3DD3C5D1A92258
2094B22551E6325D7733C41BB2F5A642ADEE417C97E0906197B5B0CD8B8D6C6B
A7A16B686D61632D736563726574F503A363616C67266373696758473045022 0
7CCAC57A1E43DF24B0847EEBF119D28DCDC5048F7DCD8EDD79E79721C41BCF2D
022100D89EC75B92CE8FF9E46FE7F8C87995694A63E5B78AB85C47B9DA
6100
```

- APDU command 3:

```
Platform Request:
80 C0 00 00 00

Authenticator Response:
1C580A8EC83A6378356381590197308201933082013 8A003020102020900859B
726CB24B4C29300A06082A8648CE3D0403023047310B3009060355040613025 5
5331143012060355040A0C0B59756269636F2054657374312230200603550 40B
0C1941757468656E74696361746F722041747465737461746F6E301E170D31
363132303431353530305A170D3236313230323131353530305A3047310B30
090603550406130255533114301206035504 0A0C0B59756269636F2054657374
312230200603550 40B0C1941757468656E74696361746F722041747465737461
74696F6E3059301306072A8648CE3D020106082A8648CE3D030107034200
61A7
```

- APDU command 4:

```
Platform Request:
80 C0 00 00 A7

Authenticator Response:
04AD11EB0E8852E53AD5DFED86B41E6134A18EC4E1AF8F221A3C7D6E636C80EA
13C3D504FF2E76211BB44525B196C44CB4849979CF6F896ECD2BB860DE1BF437
6BA30D300B30090603551D1304023000300A06082A8648CE3D04030203490030
46022100E9A39F1B03197525F7373E10CE77E78021731B94D0C03F3FDA1FD22D
B3D030E7022100C4FAEC3445A820CF43129CDB00AABEFD9AE2D874F9C5D343CB
2F113DA23723F3
9000
```

Some responses may not fit into a short APDU response. For this reason, FIDO2 authenticators MUST respond in the following way:

- If the request was encoded using *extended length* APDU encoding, the authenticator MUST respond using the extended length APDU response format.
- If the request was encoded using *short* APDU encoding, the authenticator MUST respond using ISO 7816-4 APDU chaining.

11.3.7. Commands

11.3.7.1. NFCCTAP_MSG (0x10)

The NFCCTAP_MSG command send a CTAP message to the authenticator. This command SHALL return as soon as processing is done. If the operation was not completed, it MAY return a 0x9100 result to trigger NFCCTAP_GETRESPONSE functionality if the client indicated support by setting the relevant bit in P1.

The values for P1 for the NFCCTAP_MSG command are:

P1 Bits	Meaning
0x80	The client supports NFCCTAP_GETRESPONSE
0x7F	RFU, MUST be (0x00)

Values for P2 are all RFU and MUST be set to 0.

11.3.7.2. NFCCTAP_GETRESPONSE (0x11)

The NFCCTAP_GETRESPONSE command is issued up to receiving 0x9100 unless a cancel was issued. This command SHALL return a 0x9100 result with a status indication if it has a status update, the reply to the request with a 0x9000 result code to indicate success or an error value.

All values for P1 and P2 are RFU and MUST be set to 0x00.

11.4. Bluetooth Smart / Bluetooth Low Energy Technology

See also § 11.1 Secure protocol implementation.

11.4.1. Conformance

Authenticator and client devices using Bluetooth Low Energy Technology SHALL conform to Bluetooth Core Specification 4.0 or later [BTCORE]. Bluetooth SIG specified UUID values SHALL be found on the Assigned Numbers website [BTASSNUM].

11.4.2. Pairing

Bluetooth Low Energy Technology is a long-range wireless protocol and thus has several implications for privacy, security, and overall user-experience. Because it is wireless, Bluetooth Low Energy Technology may be subject to monitoring, injection, and other network-level attacks.

For these reasons, clients and authenticators MUST create and use a long-term link key (LTK) and SHALL encrypt all communications. Authenticator MUST never use short term keys.

Because Bluetooth Low Energy Technology has poor ranging *(i.e.,* there is no good indication of proximity), it may not be clear to a FIDO client with which Bluetooth Low Energy Technology authenticator it should communicate. Pairing is the only mechanism defined in this protocol to ensure that FIDO clients are interacting with the expected Bluetooth Low Energy Technology authenticator. As a result, authenticator manufacturers SHOULD instruct users to avoid performing Bluetooth pairing in a public space such as a cafe, shop or train station.

One disadvantage of using standard Bluetooth pairing is that the pairing is "system-wide" on most operating systems. That is, if an authenticator is paired to a FIDO client which resides on an operating system where Bluetooth pairing is "system-wide", then any application on that device might be able to interact with an authenticator. This issue is discussed further in Implementation Considerations.

11.4.3. Link Security

For Bluetooth Low Energy Technology connections, the authenticator SHALL enforce `Security Mode 1, Level 2` (unauthenticated pairing with encryption) or `Security Mode 1, Level 3` (authenticated pairing with encryption) before any FIDO messages are exchanged.

11.4.4. Framing

Conceptually, framing defines an encapsulation of FIDO raw messages responsible for correct transmission of a single request and its response by the transport layer.

All requests and their responses are conceptually written as a single frame. The format of the requests and responses is given first as complete frames. Fragmentation is discussed next for each type of transport layer.

11.4.4.1. Request from Client to Authenticator

Request frames MUST have the following format

Offset	Length	Mnemonic	Description
0	1	`CMD`	Command identifier
1	1	`HLEN`	High part of data length
2	1	`LLEN`	Low part of data length
3	s	`DATA`	Data (s is equal to the length)

Supported commands are `PING`, `MSG` and `CANCEL`. The constant values for them are described below.

The `CANCEL` command cancels any outstanding `MSG` commands.

The data format for the `MSG` command is defined in § 8 Message Encoding.

11.4.4.2. Response from Authenticator to Client

Response frames MUST have the following format, which share a similar format to the request frames:

Offset	Length	Mnemonic	Description
0	1	`STAT`	Response status
1	1	`HLEN`	High part of data length

Offset	Length	Mnemonic	Description
2	1	LLEN	Low part of data length
3	s	DATA	Data (s is equal to the length)

When the status byte in the response is the same as the command byte in the request, the response is a successful response. The value ERROR indicates an error, and the response data contains an error code as a variable-length, big-endian integer. The constant value for ERROR is described below.

Note that the errors sent in this response are errors at the encapsulation layer, e.g., indicating an incorrectly formatted request, or possibly an error communicating with the authenticator's FIDO message processing layer. Errors reported by the FIDO message processing layer itself are considered a success from the encapsulation layer's point of view and are reported as a complete MSG response.

Data format is defined in § 8 Message Encoding.

11.4.4.3. Command, Status, and Error constants

The COMMAND constants and values are:

Constant	Value
PING	0x81
KEEPALIVE	0x82
MSG	0x83
CANCEL	0xbe
ERROR	0xbf

The KEEPALIVE command contains a single byte with the following possible values:

Status Constant	Value
PROCESSING	0x01
UP_NEEDED	0x02
RFU	0x00, 0x03-0xFF

The ERROR constants and values are:

Error Constant	Value	Meaning
ERR_INVALID_CMD	0x01	The command in the request is unknown/invalid
ERR_INVALID_PAR	0x02	The parameter(s) of the command is/are invalid or missing
ERR_INVALID_LEN	0x03	The length of the request is invalid
ERR_INVALID_SEQ	0x04	The sequence number is invalid
ERR_REQ_TIMEOUT	0x05	The request timed out
ERR_BUSY	0x06	The device is busy and can't accept commands at this time. The client SHOULD retry the request after a short delay. Note that the client MAY abort the transaction if the command is no longer relevant.
NA	0x0a	Value reserved (HID)
NA	0x0b	Value reserved (HID)
ERR_OTHER	0x7f	Other, unspecified error

Note: These values are identical to the HID transport values.

11.4.5. GATT Service Description

This profile defines two roles: FIDO Authenticator and FIDO Client.

- The FIDO Client SHALL be a GATT Client.
- The FIDO Authenticator SHALL be a GATT Server.

The following figure illustrates the mandatory services and characteristics that SHALL be offered by a FIDO Authenticator as part of its GATT server:

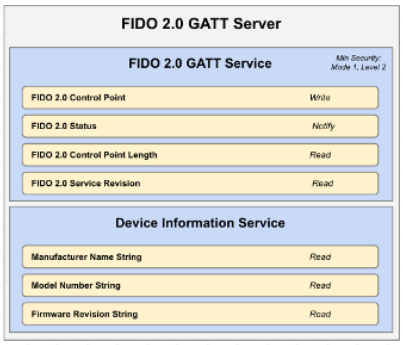

Figure 5 *Mandatory GATT services and characteristics that MUST be offered by a FIDO Authenticator. Note that the Generic Access Profile Service ([BTGAS]) is not present as it is already mandatory for any Bluetooth Low Energy Technology compliant device.*

The table below summarizes additional GATT sub-procedure requirements for a FIDO Authenticator (GATT Server) beyond those required by all GATT Servers.

GATT Sub-Procedure	Requirements
Write Characteristic Value	`Mandatory`
Notifications	`Mandatory`
Read Characteristic Descriptors	`Mandatory`
Write Characteristic Descriptors	`Mandatory`

The table below summarizes additional GATT sub-procedure requirements for a FIDO Client (GATT Client) beyond those required by all GATT Clients.

GATT Sub-Procedure	Requirements
Discover All Primary Services	`(*)`
Discover Primary Services by Service UUID	`(*)`
Discover All Characteristics of a Service	`(**)`
Discover Characteristics by UUID	`(**)`
Discover All Characteristic Descriptors	`Mandatory`
Read Characteristic Value	`Mandatory`
Write Characteristic Value	`Mandatory`
Notification	`Mandatory`
Read Characteristic Descriptors	`Mandatory`
Write Characteristic Descriptors	`Mandatory`

(*): Mandatory to support at least one of these sub-procedures. (**): Mandatory to support at least one of these sub-procedures. Other GATT sub-procedures MAY be used if supported by both client and server.

Specifics of each service are explained below. In the following descriptions: all values are big-endian coded, all strings are in UTF-8 encoding, and any characteristics not mentioned explicitly are optional.

11.4.5.1. FIDO Service

An authenticator SHALL implement the FIDO Service described below. The UUID for the FIDO GATT service is `0xFFFD`; it SHALL be declared as a Primary Service. The service contains the following characteristics:

Characteristic Name	Mnemonic	Property	Length	UUID
FIDO Control Point	`fidoControlPoint`	Write	Defined by Vendor (20-512 bytes)	F1D0FFF1-DEAA-ECEE-B42F-C9BA7ED623BB
FIDO Status	`fidoStatus`	Notify	N/A	F1D0FFF2-DEAA-ECEE-B42F-C9BA7ED623BB

Characteristic Name	Mnemonic	Property	Length	UUID
FIDO Control Point Length	`fidoControlPointLength`	Read	2 bytes	F1D0FFF3-DEAA-ECEE-B42F-C9BA7ED623BB
FIDO Service Revision Bitfield	`fidoServiceRevisionBitfield`	Read/Write	Defined by Vendor (1+ bytes)	F1D0FFF4-DEAA-ECEE-B42F-C9BA7ED623BB
FIDO Service Revision	`fidoServiceRevision`	Read	Defined by Vendor (20-512 bytes)	0x2A28

`fidoControlPoint` is a write-only command buffer.

`fidoStatus` is a notify-only response attribute. The authenticator will send a series of notifications on this attribute with a maximum length of (ATT_MTU-3) using the response frames defined above. This mechanism is used because this results in a faster transfer speed compared to a notify-read combination.

`fidoControlPointLength` defines the maximum size in bytes of a single write request to `fidoControlPoint`. This value SHALL be between 20 and 512.

`fidoServiceRevision` is superseded and is only relevant to U2F 1.0 support. It defines the revision of the U2F Service. The value is a UTF-8 string. For version 1.0 of the specification, the value `fidoServiceRevision` SHALL be `1.0` or in raw bytes: `0x312e30`. This field SHALL be omitted if protocol version 1.0 is not supported.

The `fidoServiceRevision` Characteristic MAY include a Characteristic Presentation Format descriptor with format value 0x19, `UTF-8 String`.

`fidoServiceRevisionBitfield` defines the revision of the FIDO Service. The value is a bit field which each bit representing a version. For each version bit the value is 1 if the version is supported, 0 if it is not. The length of the bitfield is 1 or more bytes. All bytes that are 0 are omitted if all the following bytes are 0 too. The byte order is big endian. The client SHALL write a value to this characteristic with exactly 1 bit set before sending any FIDO commands unless u2fServiceRevision is present and U2F 1.0 compatibility is desired. If only U2F version 1.0 is supported, this characteristic SHALL be omitted.

Byte (left to right)	Bit	Version
0	7	U2F 1.1
0	6	U2F 1.2
0	5	FIDO2
0	4-0	Reserved

For example, a device that only supports FIDO2 Rev 1 will only have a fidoServiceRevisionBitfield characteristic of length 1 with value 0x20.

11.4.5.2. Device Information Service

An authenticator SHALL implement the Device Information Service [BTDIS] and it SHOULD contain the following characteristics:

- Manufacturer Name String
- Model Number String
- Firmware Revision String

All values for the Device Information Service are left to the vendors. However, vendors SHOULD NOT create uniquely identifiable values so that authenticators do not become a method of tracking users.

11.4.5.3. Generic Access Profile Service

Every authenticator SHALL implement the Generic Access Profile Service [BTGAS] with the following characteristics:

- Device Name
- Appearance

11.4.6. Protocol Overview

The general overview of the communication protocol follows:

1. Authenticator advertises the FIDO Service.
2. Client scans for authenticator advertising the FIDO Service.
3. Client performs characteristic discovery on the authenticator.
4. If not already paired, the client and authenticator SHALL perform BLE pairing and create a LTK. Authenticator SHALL only allow connections from previously bonded clients without user intervention.
5. Client checks if the `fidoServiceRevisionBitfield` characteristic is present. If so, the client selects a supported version by writing a value with a single bit set.
6. Client reads the `fidoControlPointLength` characteristic.
7. Client registers for notifications on the `fidoStatus` characteristic.
8. Client writes a request (*e.g.*, an enroll request) into the `fidoControlPoint` characteristic.
9. Optionally, the client writes a CANCEL command to the `fidoControlPoint` characteristic to cancel the pending request.
10. Authenticator evaluates the request and responds by sending notifications over `fidoStatus` characteristic.
11. The protocol completes when either:
    - The client unregisters for notifications on the `fidoStatus` characteristic, or:
    - The connection times out and is closed by the authenticator.

11.4.7. Authenticator Advertising Format

When advertising, the authenticator SHALL advertise the FIDO service UUID.

When advertising, the authenticator MAY include the TxPower value in the advertisement (see [BTXPLAD]).

When advertising in pairing mode, the authenticator SHALL either: (1) set the LE Limited Mode bit to zero and the LE General Discoverable bit to one OR (2) set the LE Limited Mode bit to one and the LE General Discoverable bit to zero. When advertising in non-pairing mode, the authenticator SHALL set both the LE Limited Mode bit and the LE General Discoverable Mode bit to zero in the Advertising Data Flags.

The advertisement MAY also carry a device name which is distinctive and user-identifiable. For example, "ACME Key" would be an appropriate name, while "XJS4" would not be.

The authenticator SHALL also implement the Generic Access Profile [BTGAP] and Device Information Service [BTDIS], both of which also provide a user-friendly name for the device that could be used by the client.

It is not specified when or how often an authenticator should advertise, instead that flexibility is left to manufacturers.

11.4.8. Requests

Clients SHOULD make requests by connecting to the authenticator and performing a write into the `fidoControlPoint` characteristic.

Upon receiving a CANCEL request, if there is an outstanding request that can be cancelled, the authenticator MUST cancel it and that cancelled request will reply with the error CTAP2_ERR_KEEPALIVE_CANCEL. Whether a request was cancelled or not, the authenticator MUST NOT reply to the cancel message itself.

11.4.9. Responses

Authenticators SHOULD respond to clients by sending notifications on the `fidoStatus` characteristic.

Some authenticators might alert users or prompt them to complete the test of user presence (*e.g.*, via sound, light, vibration) Upon receiving any request, the authenticators SHALL send KEEPALIVE commands every `kKeepAliveMillis` milliseconds until completing processing the commands. While the authenticator is processing the request the KEEPALIVE command will contain status `PROCESSING`. If the authenticator is waiting to complete the Test of User Presence, the KEEPALIVE command will contains status `UP_NEEDED`. While waiting to complete the Test of User Presence, the authenticator MAY alert the user (e.g., by flashing) in order to prompt the user to complete the test of user presence. As soon the authenticator has completed processing and confirmed user presence, it SHALL stop sending KEEPALIVE commands, and send the reply.

Upon receiving a KEEPALIVE command, the client SHALL assume the authenticator is still processing the command; the client SHALL not resend the command. The authenticator SHALL continue sending KEEPALIVE messages at least every `kKeepAliveMillis` to indicate that it is still handling the request. Until a client-defined timeout occurs, the client SHALL NOT move on to other devices when it receives a KEEPALIVE with `UP_NEEDED` status, as it knows this is a device that can satisfy its request.

11.4.10. Framing fragmentation

A single request/response sent over Bluetooth Low Energy Technology MAY be split over multiple writes and notifications, due to the inherent limitations of Bluetooth Low Energy Technology which is not currently meant for large messages. Frames are fragmented in the following way:

A frame is divided into an *initialization fragment* and zero or more *continuation fragments*.

An initialization fragment is defined as:

Offset	Length	Mnemonic	Description
0	1	`CMD`	Command identifier
1	1	`HLEN`	High part of data length
2	1	`LLEN`	Low part of data length
3	0 to (maxLen - 3)	`DATA`	Data

where `maxLen` is the maximum packet size supported by the characteristic or notification.

In other words, the start of an initialization fragment is indicated by setting the high bit in the first byte. The subsequent two bytes indicate the total length of the frame, in big-endian order. The first `maxLen` - 3 bytes of data follow.

Continuation fragments are defined as:

Offset	Length	Mnemonic	Description
0	1	`SEQ`	Packet sequence 0x00..0x7f (high bit always cleared)
1	0 to (maxLen - 1)	`DATA`	Data

where `maxLen` is the maximum packet size supported by the characteristic or notification.

In other words, continuation fragments begin with a sequence number, beginning at 0, implicitly with the high bit cleared. The sequence number MUST wraparound to 0 after reaching the maximum sequence number of 0x7f.

Example for sending a `PING` command with 40 bytes of data with a`maxLen` of 20 bytes:

Frame	Bytes
0	`[810028] [17 bytes of data]`
1	`[00] [19 bytes of data]`
2	`[01] [4 bytes of data]`

Example for sending a ping command with 400 bytes of data with a`maxLen` of 512 bytes:

Frame	Bytes
0	`[810190] [400 bytes of data]`

11.4.11. Notifications

A client needs to register for notifications before it can receive them. Bluetooth Core Specification 4.0 or later[BT CORE] forces a device to remember the notification registration status over different connections[BTCCC]. Unless a client explicitly unregisters for notifications, the registration will be automatically restored when reconnecting. A client MAY therefor check the notification status upon connection and only register if notifications aren't already registered. Please note that some clients MAY disable notifications from a power management point of view (see below) and the notification registration is remembered per bond, not per client. A client MUST NOT remember the notification status in its own data storage.

11.4.12. Request Collisions

Because there is no concept of a session between the authenticator and a client (only between the host and the client), a BLE authenticator cannot distinguish between different clients. If two clients on the same host register for notifications from an authenticator at the same time, some existing host platforms will allow this by reusing the same underlying BLE connection. However, when the authenticator generates a notification, the host platform has insufficient information to route it to a particular client. Depending on the host platform implementation, the notification may be delivered to either or both clients. The result is undefined behavior which will likely result in both requests failing.

11.4.13. Implementation Considerations

11.4.13.1. Bluetooth pairing: Client considerations

As noted in § 11.4.2 Pairing, a disadvantage of using standard Bluetooth pairing is that the pairing is "system-wide" on most operating systems. That is, if an authenticator is paired to a FIDO client that resides on an operating system where Bluetooth pairing is "system-wide", then any application on that device might be able to interact with an authenticator. This poses both security and privacy risks to users.

While client operating system security is partly out of FIDO's scope, further revisions of this specification MAY propose mitigations for this issue.

11.4.13.2. Bluetooth pairing: Authenticator considerations

The method to put the authenticator into Pairing Mode should be such that it is not easy for the user to do accidentally **especially** if the pairing method is Just Works. For example, the action could be pressing a physically recessed button or pressing multiple buttons. A visible or audible cue that the authenticator is in Pairing Mode should be considered. As a counter example, a silent, long press of a single non-recessed button is not advised as some users naturally hold buttons down during regular operation.

Note that at times, authenticators may legitimately receive communication from an unpaired device. For example, a user attempts to use an authenticator for the first time with a new client; he turns it on, but forgets to put the authenticator into pairing mode. In this situation, after connecting to the authenticator, the client will notify the user that he needs to pair his authenticator. The authenticator should make it easy for the user to do so, e.g., by not requiring the user to wait for a timeout before being able to enable pairing mode.

Some client platforms (most notably iOS) do not expose the AD Flag LE Limited and General Discoverable Mode bits to applications. For this reason, authenticators are also strongly RECOMMENDED to include the Service Data field [BTSD] in the Scan Response. The Service Data field is 3 or more octets long. This allows the Flags field to be extended while using the minimum number of octets within the data packet. All octets that are 0x00 are not transmitted as long as all other octets after that octet are also 0x00 and it is not the first octet after the service UUID. The first 2 bytes contain the FIDO Service UUID, the following bytes are flag bytes.

To help clients show the correct UX, authenticators can use the Service Data field to specify whether or not authenticators will require a Passkey (PIN) during pairing.

Service Data Bit	Meaning (if set)
7	Device is in pairing mode.
6	Device requires Passkey Entry [BTPESTK].

11.4.14. Handling command completion

It is important for low-power devices to be able to conserve power by shutting down or switching to a lower-power state when they have satisfied a client's requests. However, the FIDO protocol makes this hard as it typically includes more than one command/response. This is especially true if a user has more than one key handle associated with an account or identity, multiple key handles may need to be tried before getting a successful outcome. Furthermore, clients that fail to send follow up commands in a timely fashion may cause the authenticator to drain its battery by staying powered up anticipating more commands.

A further consideration is to ensure that a user is not confused about which command she is confirming by completing the test of user presence. That is, if a user performs the test of user presence, that action SHOULD perform exactly one operation.

We combine these considerations into the following series of recommendations:

- Upon initial connection to an authenticator, and upon receipt of a response from an authenticator, if a client has more commands to issue, the client MUST transmit the next command or fragment within `kMaxCommandTransmitDelayMillis` milliseconds.
- Upon final response from an authenticator, if the client decides it has no more commands to send it SHOULD indicate this by disabling notifications on the `fidoStatus` characteristic. When the notifications are disabled the authenticator MAY enter a low power state or disconnect and shut down.
- Any time the client wishes to send a FIDO message, it MUST have first enabled notifications on the `fidoStatus` characteristic and wait for the ATT acknowledgement to be sure the authenticator is ready to process messages.
- Upon successful completion of a command which required a test of user presence, e.g. upon a successful authentication or registration command, the authenticator can assume the client is satisfied, and MAY reset its state or power down.

  > NOTE: authenticators supporting large blobs SHOULD wait kMaxCommandTransmitDelayMillis if the command response contained a largeBlobKey, even after consuming user presence, otherwise they may miss such commands.

- Upon sending a command response that did not consume a test of user presence, the authenticator MUST

assume that the client may wish to initiate another command and leave the connection open until the client closes it or until a timeout of at least `kErrorWaitMillis` elapses. Examples of command responses that do not consume user presence include failed authenticate or register commands, as well as get version responses, whether successful or not. After `kErrorWaitMillis` milliseconds have elapsed without further commands from a client, an authenticator MAY reset its state or power down.

Constant	Value
`kMaxCommandTransmitDelayMillis`	1500 milliseconds
`kErrorWaitMillis`	2000 milliseconds
`kKeepAliveMillis`	500 milliseconds

11.4.15. Data throughput

Bluetooth Low Energy Technology does not have particularly high throughput, this can cause noticeable latency to the user if request/responses are large. Some ways that implementers can reduce latency are:

- Support the maximum MTU size allowable by hardware (up to the 512-byte max from the Bluetooth specifications).
- Make the attestation certificate as small as possible; do not include unnecessary extensions.

11.4.16. Advertising

Though the standard does not appear to mandate it (in any way that we've found thus far), advertising and device discovery seems to work better when the authenticators advertise on all 3 advertising channels and not just one.

11.4.17. Authenticator Address Type

In order to enhance the user's privacy and specifically to guard against tracking, it is RECOMMENDED that authenticators use Resolvable Private Addresses (RPAs) instead of static addresses.

The transports that FIDO has defined are thus USB, NFC, and BLE.

11.5. Hybrid transports

Hybrid transports decouple the proof that the client platform is physically close to the authenticator or credential provider hosting device (CPHD), from the transport of CTAP2 messages between them. The hybrid transport defined here is intended to connect authenticators with cameras, typically phones, to a client platform. It involves both network communication via a service called a tunnel service, and BLE transmissions to show proximity. A **tunnel service** is a high-availability network service with a domain name known to the authenticators that use it.

11.5.1. QR-initiated Transactions

When the client platform wishes to communicate with a hybrid authenticator it may display a QR code that contains a public key and a shared secret key. The public key authenticates the client platform to any connecting authenticator and knowledge of the secret key authenticates the connecting authenticator to the client platform.

```
var (
  qrSecret [16]byte
  // The ecdsa package is used for its convenient public/private key structures,
  // but these are ECDH keys, not ECDSA.
  identityKey *ecdsa.PrivateKey
)

func showQRCode() {
  rand.Reader.Read(qrSecret[:])

  var err error
  identityKey, err = ecdsa.GenerateKey(elliptic.P256(), rand.Reader)
  if err != nil {
    panic(err)
  }
  identityKeyCompressed := compressECKey(&identityKey.PublicKey)

  printQRCode(encodeQRContents(&identityKeyCompressed, &qrSecret))
}
```

The contents of the QR code are a URI of the form FIDO:/ followed by digit-encoded data. The scheme is written in uppercase because this is more efficient in QR codes. A single foreslash follows the colon because that is

required for some devices to recognise the QR contents as a URI, but it's not a double-slash as that would indicate an authority, which this URI scheme does not use.

The encoded data is a CBOR map with integer keys mapping to key-specific values. The CBOR must be in canonical form. The keys are:

- Key 0: a 33-byte, P-256, X9.62, compressed public key.
- Key 1: a 16-byte random QR secret.
- Key 2: the number of assigned tunnel server domains known to this implementation (see `decodeTunnelServerDomain` for details).
- Key 3: (optional) the current time in epoch seconds.
- Key 4: (optional) a boolean that is true if the device displaying the QR code can perform state-assisted transactions.
- Key 5: a value from the table below, representing the user flow to follow. Implementations SHOULD treat unknown values as `ga`. This field exists so that guidance can be given to the user immediately upon scanning the QR code, prior to the credential provider hosting device/authenticator receiving any CTAP message or JSON request. While this hint SHOULD be as accurate as possible, it does not constrain the subsequent CTAP messages or JSON requests that the platform may send.

Value	Description
`ga`	getAssertion (FIDO2)
`mc`	makeCredential (FIDO2)
`dcp`	credential presentation (Digital Credentials API)
`dci`	credential issuance (Digital Credentials API)

(More fields can be added in the future as they will be ignored by older implementations.)

```
func encodeQRContents(compressedPublicKey *[33]byte, qrSecret *[16]byte) string {
  numMapElements := 6
  // GREASE QR code to ensure that keys can be added later.
  var randByte [1]byte
  rand.Reader.Read(randByte[:])
  extraKey := randByte[0]&3 == 0
  if extraKey {
    numMapElements++
  }

  var cbor []byte
  cbor = append(cbor, 0xa0+byte(numMapElements))       // CBOR map
  cbor = append(cbor, 0)                               // key 0
  cbor = append(cbor, (cborMajorByteString<<5)|24, 33) // 33 bytes
  cbor = append(cbor, compressedPublicKey[:]...)
  cbor = append(cbor, 1)                           // key 1
  cbor = append(cbor, (cborMajorByteString<<5)|16) // 16 bytes
  cbor = append(cbor, qrSecret[:]...)

  cbor = append(cbor, 2) // key 2
  n := len(assignedTunnelServerDomains)
  if n > 24 {
    panic("larger encoding needed")
  }
  cbor = append(cbor, byte(n))

  cbor = append(cbor, 3) // key 3
  cbor = append(cbor, cborEncodeInt64(time.Now().Unix())...)

  cbor = append(cbor, 4) // key 4
  cbor = append(cbor, 0xf5)  // true

  cbor = append(cbor, 5) // key 5
  cbor = append(cbor, (cborMajorByteString<<5)|2, 'm', 'c')

  if extraKey {
    cbor = append(cbor, 0x19, 0xff, 0xff, 0) // key 65535, value 0
  }

  qr := "FIDO:/" + digitEncode(cbor)
  fmt.Println(qr)
  return qr
}
```

Authenticators must use a CBOR parser to parse this information as more keys may be added in the future. The function above uses some [rfc8701] to try and ensure this.

The encoding is designed to be efficient when expressed in a QR code. Seven-byte chunks are interpreted as little-endian values and encoded as 17-digit, base 10 numbers. Any remaining bytes are encoded likewise using the minimum number of digits that some value of that number of bytes could need. Specifically, since the remainder is known to be 1, 2, 3, 4, 5, or 6 bytes long, its encoded form will take 3, 5, 8, 10, 13, or 15 digits, respectively.

```
func digitEncode(d []byte) string {
  const chunkSize = 7
  const chunkDigits = 17
  const zeros = "00000000000000000"

  var ret string
  for len(d) >= chunkSize {
    var chunk [8]byte
    copy(chunk[:], d[:chunkSize])
    v := strconv.FormatUint(binary.LittleEndian.Uint64(chunk[:]), 10)
    ret += zeros[:chunkDigits-len(v)]
    ret += v

    d = d[chunkSize:]
  }

  if len(d) != 0 {
    // partialChunkDigits is the number of digits needed to encode
    // each length of trailing data from 6 bytes down to zero. I.e.
    // it's 15, 13, 10, 8, 5, 3, 0 written in hex.
    const partialChunkDigits = 0x0fda8530

    digits := 15 & (partialChunkDigits >> (4 * len(d)))
    var chunk [8]byte
    copy(chunk[:], d)
    v := strconv.FormatUint(binary.LittleEndian.Uint64(chunk[:]), 10)
    ret += zeros[:digits-len(v)]
    ret += v
  }

  return ret
}
```

Once the QR code has been displayed the client platform awaits a connection attempt from an authenticator. This transport requires a proof of proximity to help prevent attacks, thus notification of the connection attempt comes in the form of a BLE advertisement. (Without a proof of proximity a web site could, for example, display a QR code and attempt to convince the user to scan it with their authenticator. By having the authenticator demand that the client platform prove reception of a BLE advert such an attacker would have to have control of a Bluetooth radio near to the victim.)

The UUID, `0000fff9-0000-1000-8000-00805f9b34fb`, must be included in the advert and client platforms must require that candidate devices are advertising this UUID. That UUID must also have a 20-byte service data payload which is trial decrypted to search for a match to the displayed QR code.

```
func awaitAdvert(eidKey [64]byte) [16]byte {
  // uuidsChan is a channel of UUID sets observed from some BLE device.
  // Each UUID is represented as a string in the standard format, e.g.
  // 0000fde2-0000-1000-8000-00805f9b34fb.
  var (
    serviceDataChan chan map[string][]byte
    stopScanning    func()
    err             error
  )

  if serviceDataChan, stopScanning, err = bleScanForServiceData(); err != nil {
    panic(err)
  }
  defer stopScanning()

  const UUID = "0000fff9-0000-1000-8000-00805f9b34fb"

  for serviceData := range serviceDataChan {
    cableData, ok := serviceData[UUID]
    if !ok {
      continue
    }

    if payload, ok := trialDecrypt(&eidKey, cableData); ok {
      return payload
    }
  }

  panic("UUID channel closed")
}
```

In order to derive the key needed to trial decrypt BLE adverts, the following key derivation is used. Whenever a key is needed for a specific purpose it is always derived from a parent key in order to ensure domain separation. The derivation uses [RFC5869] with SHA-256, where the input keying material is the parent key, the salt is an optional input, and the info value is a 32-bit, little-endian, purpose identifier.

```
type keyPurpose uint32

const (
  keyPurposeEIDKey   keyPurpose = 1
  keyPurposeTunnelID keyPurpose = 2
  keyPurposePSK      keyPurpose = 3
)

func derive(output, secret, salt []byte, purpose keyPurpose) {
  if uint32(purpose) >= 0x100 {
    panic("unsupported purpose")
  }

  var purpose32 [4]byte
  purpose32[0] = byte(purpose)

  h := hkdf.New(sha256.New, secret, salt, purpose32[:])
  if n, err := h.Read(output); err != nil || n != len(output) {
    panic("HKDF error")
  }
}
```

The key used to decrypt adverts is then a 64-byte value derived from the QR secret with `keyPurposeEIDKey`. The term “EID” is historical and does not stand for anything here.

```
func awaitQRAdvert() [16]byte {
  var eidKey [32 + 32]byte
  derive(eidKey[:], qrSecret[:], nil, keyPurposeEIDKey)
  return awaitAdvert(eidKey)
}
```

When decrypting adverts, these 64 bytes of EID key are considered as a pair of 256-bit keys where the first 32 bytes are an AES key and the second 32 bytes are an HMAC-SHA256 key. A candidate BLE advert is valid if the final four bytes are a correct HMAC tag of the other 16 bytes. For each valid BLE advert, those initial 16 bytes are then taken to be an AES block and decrypted with the AES key.

This is a poor-man’s substitute for a wide-block mode, but wide-block modes are non-standard. There is no more space in the BLE advert so a nonce cannot be included. Since it’s possible that two authenticators could scan the same QR code and broadcast based on the same key, avoiding a mode that XORs plaintext with a keystream avoids potential complications.

```
func trialDecrypt(eidKey *[64]byte, candidateAdvert []byte) (plaintext [16]byte, ok bool) {
  var zeros [16]byte
  if len(candidateAdvert) != 20 {
    return zeros, false
  }

  h := hmac.New(sha256.New, eidKey[32:])
  h.Write(candidateAdvert[:16])
  expectedTag := h.Sum(nil)

  if !hmac.Equal(expectedTag[:4], candidateAdvert[16:]) {
    return zeros, false
  }

  block, err := aes.NewCipher(eidKey[:32])
  if err != nil {
    panic(err)
  }

  block.Decrypt(plaintext[:], candidateAdvert[:16])
  if !reservedBitsAreZero(plaintext) {
    return zeros, false
  }

  return plaintext, true
}
```

Once successfully authenticated and decrypted, a BLE advert yields 16 bytes of plaintext. These 16 bytes consist of (in order):

- A flags byte, which is currently zero. This could be used for versioning in the future.
- 80 bits of connection nonce.

- A 24-bit routing ID.
- A 16-bit tunnel service identifier.

```
func reservedBitsAreZero(plaintext [16]byte) bool {
  return plaintext[0] == 0
}

func unpackDecryptedAdvert(plaintext [16]byte) (
  nonce [10]byte,
  routingID [3]byte,
  encodedTunnelServerDomain uint16) {

  copy(nonce[:], plaintext[1:])
  copy(routingID[:], plaintext[11:])
  encodedTunnelServerDomain = uint16(plaintext[14]) | (uint16(plaintext[15]) << 8)
  return
}
```

The connection nonce is the value that demonstrates possession of the BLE advert, and thus proximity to the authenticator.

The tunnel service relays messages to and from the authenticator. It is a property of the authenticator because, as detailed later, it can contact the authenticator on request when a client platform is “linked”. The protocol between the authenticator and the tunnel service, and details about how the service later contacts the authenticator, are a private detail of the authenticator's implementation.

The encoded tunnel service identifier is a uint16. Values zero through 255 are assigned, and values >= 256 are translated into a domain name by hashing. A “cable” label is prepended to hashed domains to allow for use of CNAME records.

Domains are assigned sequentially and the number of assigned domains is included in the QR code. Therefore authenticators can know whether a peer will recognise an assigned domain or not and can potentially fall back to a hashed domain for compatibility.

These are the currently assigned domains, in order:

```
var assignedTunnelServerDomains = []string{"cable.ua5v.com", "cable.auth.com"}

func decodeTunnelServerDomain(encoded uint16) (string, bool) {
  if encoded < 256 {
    if int(encoded) >= len(assignedTunnelServerDomains) {
      return "", false
    }
    return assignedTunnelServerDomains[encoded], true
  }

  shaInput := []byte{
    0x63, 0x61, 0x42, 0x4c, 0x45, 0x76, 0x32, 0x20,
    0x74, 0x75, 0x6e, 0x6e, 0x65, 0x6c, 0x20, 0x73,
    0x65, 0x72, 0x76, 0x65, 0x72, 0x20, 0x64, 0x6f,
    0x6d, 0x61, 0x69, 0x6e,
  }
  shaInput = append(shaInput, byte(encoded), byte(encoded>>8), 0)
  digest := sha256.Sum256(shaInput)

  v := binary.LittleEndian.Uint64(digest[:8])
  tldIndex := uint(v & 3)
  v >>= 2

  ret := "cable."
  const base32Chars = "abcdefghijklmnopqrstuvwxyz234567"
  for v != 0 {
    ret += string(base32Chars[v&31])
    v >>= 5
  }

  tlds := []string{".com", ".org", ".net", ".info"}
  ret += tlds[tldIndex&3]

  return ret, true
}
```

The routing ID is an opaque value that must be provided to the tunnel service and which aids its operation.

The client platform is now in possession of everything needed to establish the tunnel to the authenticator. The first step of doing so is to derive the tunnel ID, a 128-bit identifier that the tunnel service recognises and which identifies the exchange separate from any others that the tunnel service might concurrently be facilitating. It is derived, as detailed above, from the QR secret. It is not dependent on the nonce from the BLE advert because

that would mean that the tunnel service could try and brute-force the nonce from the tunnel ID. The tunnel service is trusted by the authenticator, but no need to trust it more than necessary.

With the tunnel ID in hand, the tunnel service is contacted viaWebSockets. In order to request a connection to a given tunnel ID, the path of the WebSockets URL is set to `/cable/connect/` followed by the lower-case, hex-encoded routing ID, another foreslash, then the lower-case, hex-encoded tunnel ID. The WebSocket connection must set the subprotocol identifier to `fido.cable`.

Implementations must follow HTTP redirects when establishing the WebSocket connection.

```
var tunnelServerDomain string

const subprotocol = "fido.cable"

func connectToPhone(advertPlaintext [16]byte) {
  _, routingID, encodedTunnelServerDomain := unpackDecryptedAdvert(advertPlaintext)

  var ok bool
  if tunnelServerDomain, ok = decodeTunnelServerDomain(encodedTunnelServerDomain); !ok {
    panic("unknown tunnel server domain")
  }

  var tunnelID [16]byte
  derive(tunnelID[:], qrSecret[:], nil, keyPurposeTunnelID)

  connectURL := "wss://" +
    tunnelServerDomain +
    "/cable/connect/" +
    hex.EncodeToString(routingID[:]) +
    "/" +
    hex.EncodeToString(tunnelID[:])

  conn, _, err := (&websocket.Dialer{
    Subprotocols: []string{subprotocol},
  }).Dial(connectURL, nil)

  if err != nil {
    panic(err)
  }

  if conn.Subprotocol() != subprotocol {
    panic("tunnel service picked wrong subprotocol")
  }

  doQRHandshake(conn, advertPlaintext)
}
```

With the tunnel established, messages are exchanged in binary WebSocket frames and no other frame types are permitted on the connection. The authenticator and client platform first perform a cryptographic handshake to establish a forward-secure, authenticated connection. This handshake is Noise KNpsk0 using P-256, SHA-256, and AES-256-GCM.

The client platform speaks first to prove possession of the BLE advert. Theauthenticator thus needs only to receive the client platform's handshake message and send a reply in order to complete the handshake. The KNpsk0 pattern requires that the initiator (the client platform) have shared a public key in advance with the responder (the authenticator), and that both sides share a symmetric key. The pre-exchanged public key was passed to the authenticator in the QR code, and the pre-shared symmetric key is derived from the QR secret and decrypted BLE advert. (The full BLE advert is included in the PSK derivation to ensure that any future additions to the advert format are automatically authenticated.)

```
func doQRHandshake(websocketConn *websocket.Conn, advertPlaintext [16]byte) {
  var psk [32]byte
  derive(psk[:], qrSecret[:], advertPlaintext[:], keyPurposePSK)

  conn, handshakeHash := doHandshake(websocketConn, psk, identityKey, nil)
  readPostHandshakeMessage(conn, handshakeHash)
}

func doHandshake(websocketConn *websocket.Conn,
  psk [32]byte,
  identityKey *ecdsa.PrivateKey,
  // peerIdentity is not used until linked connections are discussed, below.
  peerIdentity *ecdsa.PublicKey) (

  conn io.ReadWriteCloser,
  handshakeHash [32]byte) {

  msg, ephemeralKey, noiseState := initialHandshakeMessage(psk, identityKey, peerIdentity)
  if err := websocketConn.WriteMessage(websocket.BinaryMessage, msg); err != nil {
    panic(err)
  }

  msgType, handshakeMessageFromPhone, err := websocketConn.ReadMessage()
  if err != nil {
    panic(err)
  }
  if msgType != websocket.BinaryMessage {
    panic("non-binary message received on WebSocket")
  }

  trafficKeys, handshakeHash := processHandshakeResponse(
    handshakeMessageFromPhone, ephemeralKey, identityKey, noiseState)

  conn = newCableConn(&websocketAdaptor{websocketConn}, trafficKeys)
  return conn, handshakeHash
}
```

As referenced above, the handshake itself is Noise NKpsk0. The following functions implement both NKpsk0 and KNpsk0 because the latter will be needed below. The underlying Noise operations are specified in the Noise specification. p256X962Length is the length of an uncompressed, X9.62, P-256 point, in bytes.

```
const p256X962Length = 1 + 32 + 32

func initialHandshakeMessage(
  psk [32]byte,
  priv *ecdsa.PrivateKey,
  peerPub *ecdsa.PublicKey) (

  msg []byte,
  ephemeralKey *ecdsa.PrivateKey,
  noise *noiseState) {

  if (priv == nil) == (peerPub == nil) {
    panic("exactly one of priv and peerPub must be given")
  }

  var ns *noiseState
  if peerPub != nil {
    ns = newNoise(noiseNKpsk0)
    ns.mixHash([]byte{0})
    ns.mixHashPoint(peerPub)
  } else {
    ns = newNoise(noiseKNpsk0)
    ns.mixHash([]byte{1})
    ns.mixHashPoint(&priv.PublicKey)
  }

  ns.mixKeyAndHash(psk[:])

  ephemeralKey, err := ecdsa.GenerateKey(elliptic.P256(), rand.Reader)
  if err != nil {
    panic(err)
  }

  ephemeralKeyBytes := elliptic.Marshal(ephemeralKey.Curve, ephemeralKey.X, ephemeralKey.Y)
  ns.mixHash(ephemeralKeyBytes)
  ns.mixKey(ephemeralKeyBytes)

  if peerPub != nil {
    ns.mixKey(ecdh(ephemeralKey, peerPub.X, peerPub.Y))
  }
```

```
  msg = append(msg, ephemeralKeyBytes...)
  msg = append(msg, ns.encryptAndHash(nil)...)

  return msg, ephemeralKey, ns
}

func processHandshakeResponse(
  peerHandshakeMessage []byte,
  ephemeralKey *ecdsa.PrivateKey,
  priv *ecdsa.PrivateKey,
  ns *noiseState) (

  keys trafficKeys,
  handshakeHash [32]byte) {

  if len(peerHandshakeMessage) < p256X962Length {
    panic("handshake too short")
  }

  peerPointBytes := peerHandshakeMessage[:p256X962Length]
  ciphertext := peerHandshakeMessage[p256X962Length:]

  ns.mixHash(peerPointBytes)
  ns.mixKey(peerPointBytes)

  peerPointX, peerPointY := elliptic.Unmarshal(ephemeralKey.Curve, peerPointBytes)
  if peerPointX == nil {
    panic("peer's point is not on the curve")
  }

  ns.mixKey(ecdh(ephemeralKey, peerPointX, peerPointY))

  if priv != nil {
    ns.mixKey(ecdh(priv, peerPointX, peerPointY))
  }

  plaintext, ok := ns.decryptAndHash(ciphertext)
  if !ok || len(plaintext) != 0 {
    panic("bad handshake")
  }

  return ns.split(), ns.handshakeHash()
}
```

Once the handshake is complete, the traffic-keys that result from Noise's`Split` operation are assigned to the client platform-to-authenticator and authenticator-to-client platform flows, respectively. Future messages on the tunnel are padded and AES-256-GCM encrypted. Padding is performed by setting the final byte of the plaintext to the number of preceding bytes that are padding. Padding bytes can take any value but zero is recommended. Implementations can use a padding granularity up to 256 bytes, but 32 is recommended. Nonces are per-direction counters, big-endian encoded into 12 bytes. The additional data for every message is empty.

Implementations may terminate connections that exceed 24 bits of nonce to avoid worrying about nonce overflow.

```
type cableConn struct {
  conn              io.ReadWriteCloser
  readKey, writeKey [32]byte
  readSeq, writeSeq uint32
}

var additionalData []byte = nil

func setupAEAD(counter *uint32, key *[32]byte) (nonce [12]byte, aead cipher.AEAD) {
  if *counter > 1<<24 {
    // To avoid dealing with the nonce counter overflowing,
    // connections are capped at 2^24 messages.
    panic("too many messages")
  }

  binary.BigEndian.PutUint32(nonce[8:], *counter)
  *counter++

  block, err := aes.NewCipher(key[:])
  if err != nil {
    panic(err)
  }
  if aead, err = cipher.NewGCM(block); err != nil {
    panic(err)
  }

  return
}

func (c *cableConn) Write(msg []byte) (int, error) {
  const paddingGranularity = 32
  if len(msg) > 1<<20 {
    // 1MiB is comfortably larger than any valid CTAP2 message and
    // this limit moots possible overflows below.
    panic("plaintext too large")
  }

  extraBytes := paddingGranularity - (len(msg) % paddingGranularity)
  paddedLen := len(msg) + extraBytes

  paddedMsg := make([]byte, paddedLen, paddedLen)
  copy(paddedMsg, msg)
  paddedMsg[len(paddedMsg)-1] = byte(extraBytes) - 1

  nonce, aead := setupAEAD(&c.writeSeq, &c.writeKey)
  ciphertext := aead.Seal(paddedMsg[:0], nonce[:], paddedMsg, additionalData)

  if n, err := c.conn.Write(ciphertext); err != nil {
    return 0, err
  } else if n != len(ciphertext) {
    return 0, errors.New("unexpected short write")
  } else {
    return len(msg), nil
  }
}
```

Decryption consists of the reverse of the encryption steps:

```
func (c *cableConn) Read(buf []byte) (int, error) {
  n, err := c.conn.Read(buf)
  if err != nil {
    return n, err
  }
  buf = buf[:n]

  nonce, aead := setupAEAD(&c.readSeq, &c.readKey)
  plaintext, err := aead.Open(buf[:0], nonce[:], buf, additionalData)
  if err != nil {
    panic("decryption failure")
  }

  if len(plaintext) == 0 {
    panic("invalid message")
  }
  paddingBytes := int(plaintext[len(plaintext)-1])
  if paddingBytes+1 > len(plaintext) {
    panic("invalid message")
  }

  plaintext = plaintext[:len(plaintext)-1-paddingBytes]
  if len(plaintext) > len(buf) {
    panic("message too large")
  }
  n = copy(buf, plaintext)
  return n, nil
}
```

The first message from the authenticator is the “post handshake” message. This message contains the authenticator’s getInfo response, to save a round-trip. This message contains a CBOR map, which must be in CTAP2 canonical form.

The CBOR map contains the following:

- Key 0: (optional) a bytestring containing only zero bytes, for padding.
- Key 1: the getInfo response, a bytestring.
- Key 2: reserved.
- Key 3: (optional) an array of strings representing supported features, as defined in the table below. The absense of this key MUST be treated as if it were present with the value ["ctap"].

Value	Description
ctap	The credential provider hosting device (CPHD) supports CTAP2 requests.
dc	The credential provider hosting device (CPHD) supports Digital Credentials requests usingJSON-based messages.

```
type postHandshakeMessage struct {
  GetInfoReply []byte   \`cbor:"1"\`
}

func readPostHandshakeMessage(conn io.ReadWriteCloser, handshakeHash [32]byte) {
  msgBytes := make([]byte, 128<<10)
  n, err := conn.Read(msgBytes)
  if err != nil {
    panic("read failure: " + err.Error())
  }

  var msg postHandshakeMessage
  if !cborParse(&msg, msgBytes[:n]) {
    fmt.Printf("%x\n", msgBytes)
    panic("invalid post-handshake message")
  }

  if msg.GetInfoReply == nil {
    panic("post-handshake message is missing getInfo response")
  }

  sendCTAP2Request(conn, handshakeHash)
}
```

With the tunnel now fully set up, the parties can exchange messages. Each message begins with a byte that denotes the type of the message. An empty message is thus a protocol error. The following types are defined:

- 0: a shutdown message.
- 1: a CTAP message.

- 2: an update message.
- 3: a JSON-based message.

A shutdown message may only be sent by the client to the authenticator. The message must consist only of the type byte. It indicates that the client will not send any further CTAP commands to the authenticator. The authenticator may choose to close the connection upon receiving such a message. If it supports state-assisted transactions then the client SHOULD accept messages from the authenticator for at least two minutes after sending a shutdown message.

A CTAP message contains a CTAP2 payload for processing. For example, when sent from client to authenticator, the bytes following the type byte will be a CTAP2 command.

An update message may be sent by either side at any time. The bytes following the type byte must be a CBOR map encoded using the canonical rules. Unknown keys in the map must be ignored. The codespace of keys is separate for each direction. Currently keys are only defined in the authenticator to client direction:

- Key 0: (optional) a bytestring containing only zero bytes, for padding.
- Key 1: (optional) a map containing linking information.

The linking map contains:

- Key 1: the “contact ID”, an opaque value that can be presented to the tunnel service to identify this authenticator. (For Android this an FCM registration token.)
- Key 2: the “link ID”, an opaque value that identifies this link to theauthenticator. This must be sent back to the authenticator when contacting it so that it knows what set of keys to use for thisclient platform.
- Key 3: the “link secret”, a shared secret key.
- Key 4: the authenticator's public key, X9.62 uncompressed. This value is global to theauthenticator and identifies it. If the same authenticator is used multiple times with a a QR-initiated transaction then this lets the client platform deduplicate the linking information. Desktops may sync linking information using systems like Chrome Sync and this public key prevents a client platform with linking information from impersonating the authenticator to another client platform.
- Key 5: the authenticator's name, for the purposes of identifying it to the user. For example “Pixel 3 XL”.
- Key 6: the handshake signature. See below.

```
type authenticatorToClientUpdateMessage struct {
  LinkingData linkData   \`cbor:"1"\`
}

type linkData struct {
  ContactID              []byte   \`cbor:"1"\`
  LinkID                 [8]byte  \`cbor:"2"\`
  LinkSecret             [32]byte \`cbor:"3"\`
  AuthenticatorPublicKey [65]byte \`cbor:"4"\`
  AuthenticatorName      string   \`cbor:"5"\`
  Signature              [32]byte \`cbor:"6"\`

  authPublicKey      *ecdsa.PublicKey
  tunnelServerDomain string
}

var initialLinkData *linkData

func parseUpdateMessage(payload []byte, handshakeHash [32]byte) {
  var msg authenticatorToClientUpdateMessage
  if !cborParse(&msg, payload) {
    fmt.Printf("%x\n", payload)
    panic("invalid update message")
  }

  // Linking data is optional.
  if msg.LinkingData.ContactID == nil {
    return
  }

  initialLinkData = &msg.LinkingData

  pubKey := &ecdsa.PublicKey{
    Curve: elliptic.P256(),
  }
  pubKey.X, pubKey.Y = elliptic.Unmarshal(pubKey.Curve, initialLinkData.AuthenticatorPublicK
ey[:])
  if pubKey.X == nil {
    panic("bad link public key")
  }

  if !verifySignature(initialLinkData.Signature, handshakeHash, pubKey) {
    panic("invalid link signature")
  }

  initialLinkData.tunnelServerDomain = tunnelServerDomain
  initialLinkData.authPublicKey = pubKey

  fmt.Printf("Linking information received\n")
}
```

The signature in the linking data serves to prove possession of the claimed public key. This is needed because that public key is an identifier and future linking messages that claim the same public key will replace older ones. This allows a authenticator to update its linking information at theclient platform, but authenticators should not be able to replace another authenticator's data.

The handshake hash is Noise's channel binding value and hashes the handshake transcript. Since the authenticator's public key is used as an ECDH key in later Nosie handshakes, we don't want to overload it as an ECDSA key too. Thus the "signature" in the linking message is actually an HMAC of the handshake hash under the shared key between the authenticator's key and the key in theclient platform's QR code.

```
func verifySignature(sig, handshakeHash [32]byte, pubKey *ecdsa.PublicKey) bool {
  sharedKey := ecdh(identityKey, pubKey.X, pubKey.Y)
  h := hmac.New(sha256.New, sharedKey)
  h.Write(handshakeHash[:])
  expectedSignature := h.Sum(nil)
  return hmac.Equal(expectedSignature, sig[:])
}
```

The client platform must send CTAP2 commands in order to direct the authenticator to perform some action. Typically in a CTAP2 exchange that would be a getInfo request. However, since the response was already provided in the post-handshake message, the client platform can immediately send a more substantial request. The example below sends a superfluous `authenticatorGetInfo` request.

```
const (
  typeShutdown = 0
  typeCTAP = 1
  typeUpdate = 2
  typeJSON = 3
)

func sendCTAP2Request(conn io.ReadWriteCloser, handshakeHash [32]byte) {
  authenticatorGetInfoRequest := []byte{typeCTAP, 4}
  if _, err := conn.Write(authenticatorGetInfoRequest); err != nil {
    panic("write failed")
  }

  for {
    reply := make([]byte, 128<<10)
    n, err := conn.Read(reply)
    if err != nil {
      fmt.Printf("WebSocket closed\n");
      return
    }
    reply = reply[:n]

    if len(reply) == 0 {
      panic("invalid empty message received")
    }

    msgType, reply := reply[0], reply[1:]

    switch msgType {
    case typeShutdown:
      panic("shutdown message received from authenticator")

    case typeCTAP:
      fmt.Printf("CTAP reply: %x\n", reply)
      if _, err := conn.Write([]byte{typeShutdown}); err != nil {
        panic("write failed")
      }

    case typeUpdate:
      parseUpdateMessage(reply, handshakeHash)

    default:
      panic("invalid message type received")
    }
  }

  conn.Close()
}
```

11.5.2. State-assisted Transactions

If a client platform has linking information for a authenticator, from a previous QR-initiated transaction, then it doesn't need to show a QR code in order to contact that authenticator again. By making a WebSockets connection to the cached tunnel service with the path `/cable/contact/` followed by the base64url-encoded contact ID, the tunnel service will attempt to establish a tunnel with the identified authenticator. If the tunnel service believes that the authenticator is permanently uncontactable (e.g. because the user opted to unlink this client platform on the authenticator) then the tunnel server returns HTTP status 410 and the client platform should forget the link information.

The authenticator needs two values to start communicating on the tunnel: the link ID so that it knows which client platform is contacting it (and thus which keys to use), and a nonce from the client platform. The latter diversifies the key that encrypts the BLE advert and prevents anyone passively listening from being able to link the advert to any set of link keys retrospectively. The two values are called the "client payload" and are hex-encoded in a `X-caBLE-Client-Payload` HTTP header.

In order to aid the authenticator in displaying UI to the user, a third value is encoded in the client payload: a hint about whether the following transaction will be a makeCredential or a getAssertion.

Once the tunnel is ready the authenticator will send its handshake message and start advertising over BLE as a proximity challenge. The BLE advert in this case contains the same initial flags byte, which must be zero, and the remaining 15 bytes are all nonce. Once the BLE advert is received, the client platform can calculate the handshake PSK and respond.

The handshake in this case will be NKpsk0 because now it is the authenticator that has previously shared a public key.

```
func performStateAssistedConnection(linkData *linkData) {
  contactURL := "wss://" +
    linkData.tunnelServerDomain +
    "/cable/contact/" +
    base64.RawURLEncoding.EncodeToString(linkData.ContactID)

  clientNonce, clientPayload := constructClientPayload(linkData)
  headers := make(http.Header)
  headers.Add("X-caBLE-Client-Payload", hex.EncodeToString(clientPayload))

  websocketConn, resp, err := (&websocket.Dialer{
    Subprotocols: []string{subprotocol},
  }).Dial(contactURL, headers)

  if err != nil {
    if resp != nil && resp.StatusCode == 410 {
      panic("device unlinked")
    }
    panic(err)
  }

  if websocketConn.Subprotocol() != subprotocol {
    panic("tunnel service picked wrong subprotocol")
  }

  var eidKey [64]byte
  derive(eidKey[:], linkData.LinkSecret[:], clientNonce[:], keyPurposeEIDKey)

  println("waiting for advert")
  advertPlaintext := awaitAdvert(eidKey)
  println("have advert")
  if !reservedBitsAreZero(advertPlaintext) {
    panic("bad link advert")
  }

  var psk [32]byte
  derive(psk[:], linkData.LinkSecret[:], advertPlaintext[:], keyPurposePSK)

  doHandshake(websocketConn, psk, nil, linkData.authPublicKey)
  println("State-assisted connection complete")
}
```

The client payload is encoded in a CBOR message (which must follow the encoding rules) using the following format:

- Key 1: the 8-byte link ID; a bytestring.
- Key 2: a 16-byte nonce generated by the client platform; a bytestring.
- Key 3: a value from the table below, representing the user flow to follow.

Value	Description
`ga`	getAssertion (FIDO2)
`mc`	makeCredential (FIDO2)
`dcp`	credential presentation (Digital Credentials)
`dci`	credential issuance (Digital Credentials)

```
func constructClientPayload(linkData *linkData) (nonce [16]byte, payload []byte) {
  rand.Reader.Read(nonce[:])

  payload = append(payload, 0xa3)                    // Three-element CBOR map
  payload = append(payload, 1)                       // key 1
  payload = append(payload, cborMajorByteString<<5|8) // 8 bytes
  payload = append(payload, linkData.LinkID[:]...)
  payload = append(payload, 2)                        // key 2
  payload = append(payload, cborMajorByteString<<5|16) // 16 bytes
  payload = append(payload, nonce[:]...)
  payload = append(payload, 3)                   // key 3
  payload = append(payload, cborMajorString<<5|2) // two-byte string
  payload = append(payload, 'g', 'a')            // getAssertion

  return nonce, payload
}
```

From this point, the connection works the same as the QR-initiated one. The authenticator can optionally send linking information in the post-handshake message if it wishes to update any linking information and then CTAP2 messages flow as before.

12. Defined Extensions§

This section defines authenticator extensions and any necessary corresponding client extension processing for them.

> NOTE: extensions may be defined such that extension processing may occur without any extension input.

12.1. Credential Protection (credProtect)§

12.1.1. Feature detection§

Extension identifier

`credProtect`

This registration extension allows relying parties to specify a credential protection policy when creating a credential. Additionally, authenticators MAY choose to establish a default credential protection policy greater than userVerificationOptional (the lowest level) and unilaterally enforce such policy. Authenticators not supporting some form of user verification MUST NOT support this extension.

Authenticators supporting some form of user verification MUST process this extension and persist the credProtect value with the credential, even if the authenticator is notprotected by some form of user verification at the time.

> NOTE: support for this extension is mandatory in some cases. See§ 9 Mandatory features.

Client extension input

`create()` : A single USVString specifying a protection level of the credential to be created.

```
partial dictionary AuthenticationExtensionsClientInputs {
  USVString credentialProtectionPolicy;
  boolean enforceCredentialProtectionPolicy = false;
};
```

Client extension processing

If this extension is not present in anauthenticatorMakeCredential request:

1. The platform MAY enforce its own default`credentialProtectionPolicy` value by adding this extension.

If this extension is present in anauthenticatorMakeCredential request:

1. Verify that the `credentialProtectionPolicy` string value is one of following:
   - **userVerificationOptional**:
     - This reflects "FIDO_2_0" semantics. In this configuration, performingsome form of user verification is OPTIONAL with or without credentialID list. This is the default state of the credential if the extension is not specified.
   - **userVerificationOptionalWithCredentialIDList**:
     - In this configuration, credential is discovered only when itscredentialID is provided by the platform or when some form of user verification is performed.
   - **userVerificationRequired**:
     - This reflects that discovery and usage of the credential MUST be preceded bysome form of user verification.
2. Evaluate the boolean `enforceCredentialProtectionPolicy`'s value. This controls whether it is better to fail to create a credential rather than ignore the protection policy. When `enforceCredentialProtectionPolicy` is true, and `credentialProtectionPolicy`'s value is either *userVerificationOptionalWithCredentialIDList* or *userVerificationRequired*, the platform SHOULD NOT create the credential in a way that does not implement the requested protection policy. (For example, by creating it on an authenticator that does not support this extension.)

   The platform SHOULD NOT alter the `credentialProtectionPolicy` value: the Relying Party's desired credential protection policy overrides any default credential protection policies imposed by the platform.

> NOTE: Platforms may require enterprise policy, or other configuration to conform to standards like[FIPS140-3]. Those may require modification of the Relying Party's desired credential protection policy. The Relying Party's desired credential protection policy SHOULD NOT be modified in other circumstances.

> NOTE: For non-discoverable credentials, `credentialProtectionPolicy` values *userVerificationOptional* and *userVerificationOptionalWithCredentialIDList* will both have the same authenticator behaviour since the Relying Party must always supply an allowList containing credential IDs when attempting to use authenticatorGetAssertion with such credentials.

Client extension output

None. Authenticator returns the result in authenticator extension output.

Authenticator extension input

Map credentialProtectionPolicy value to credProtect and send it to the authenticator.

- **authenticatorMakeCredential additional behaviours**

The list of possible values for credProtect is:

credentialProtectionPolicy	credProtect Value
userVerificationOptional	0x01
userVerificationOptionalWithCredentialIDList	0x02
userVerificationRequired	0x03

The platform sends the authenticatorMakeCredential request with the following CBOR map entry in the "extensions" field to the authenticator:

- "credProtect": `<credProtect Value>`

The value of the map entry MUST be the credProtect value the authenticator set for the created credential.

NOTE: Some authenticators for high-security environments may be configured to always set credProtect 3 for all created credentials regardless of what the platform requests. In this case if a Relying Party causes an authenticatorMakeCredential request to be sent with credProtect 2 (using thecredProtect extension), the authenticator will create the credential, set the credential's credProtect policy to 3, and respond via the credProtect extension result that it set the policy to 3.

EXAMPLE 8

Sample CTAP2 authenticatorMakeCredential Request (CBOR):

```
{
  ...
  6: {"credProtect": 0x01},
  ...
}
```

Authenticator extension processing

credProtect value is persisted with the credential. If no credProtect extension was included in the request the authenticator SHOULD use the default value of 1 for compatibility with CTAP2.0 platforms. The authenticator MUST NOT return an unsolicited credProtect extension output.

Authenticator extension output

- The authenticator responds with the following CBOR map entry in the "extensions" field of the authenticator data object:
  - "credProtect": `<credProtect Value>`

EXAMPLE 9

Sample "extensions" field value in the authenticatorData:

```
{"credProtect": 0x01}
```

12.2. Credential Blob (credBlob)

This extension enables RPs to provide a small amount of extra credential configuration information (**credBlob value**) to the authenticator when a credential is made. This information is an opaque blob to the authenticator. Authenticator MUST support at least 32 bytes to be stored. Authenticator reflects amount of byte storage it supports as `maxCredBlobLength` parameter in `authenticatorGetInfo`. If authenticator supports this extension,

1. If the rk option ID is present and true
   - Authenticator MUST support it fordiscoverable credentials.
   - Authenticator MAY choose to also support it fornon-discoverable credentials.
2. Else (implying the authenticator does not supportdiscoverable credentials)
   - Authenticator MUST support it fornon-discoverable credentials.

If RPs want to put PII or sensitive information in this field, they MUST use thecredProtect extension, setting the `credentialProtectionPolicy` as `userVerificationRequired` and `enforceCredentialProtectionPolicy` as `true`. This will prevent a credential that is not protected bysome form of user verification from being created.

Authenticators MUST support `credProtect` extension if they wish to support `credBlob` extension.

12.2.1. Feature detection

To detect whether authenticator supports this feature, following conditions MUST be met:

- Authenticator MUST return `credBlob` in extensions field in authenticatorGetInfo in addition to other extensions it may support.
  - Authenticator MUST also support dependent extension `credProtect`.
- Authenticator MUST return `maxCredBlobLength (0x0F)` in authenticatorGetInfo.

Extension identifier
`credBlob`

Client extension input
`create()` : ArrayBuffer containing opaque data in an RP-specific format.

```
partial dictionary AuthenticationExtensionsClientInputs {
  ArrayBuffer credBlob;
};
```

`get()` : A boolean value to indicate that this extension is requested by the Relying Party.

```
partial dictionary AuthenticationExtensionsClientInputs {
  boolean getCredBlob;
};
```

Client extension processing
`create()` : If credBlob size is less than or equal to maxCredBlobLength, platform passes the information to the authenticator. Otherwise, platform ignores it.

`get()` : None.

Client extension output
`create()` : Boolean indicating whether the requested blob was stored, mirroring the authenticator's output.

```
partial dictionary AuthenticationExtensionsClientOutputs {
  boolean credBlob;
};
```

`get()` : ArrayBuffer containing the requested blob, or empty if none was found, mirroring the authenticator's output.

```
partial dictionary AuthenticationExtensionsClientOutputs {
  ArrayBuffer getCredBlob;
};
```

Authenticator extension input

- **authenticatorMakeCredential authenticator extension input**
  - The platform sends the credBlob value in authenticatorMakeCredential request with the following CBOR map entry in the "extensions" field to the authenticator:
    - "credBlob": Byte String containing the credBlob value
- **authenticatorGetAssertion authenticator extension input**
  - The platform sends the authenticatorGetAssertion request with the following CBOR map entry in the "extensions" field to the authenticator:
    - "credBlob":true

Authenticator extension processing
credBlob value is persisted with the Credential during authenticatorMakeCredential and returned during authenticatorGetAssertion.

Authenticator extension output

- **authenticatorMakeCredential authenticator extension output**
  - If the authenticator is able to store the credBlob value, it returns the following CBOR map entry in the "extensions" fields to the authenticator:
    - "credBlob": true
  - If the authenticator is not able to store the credBlob value (e.g. credBlob exceeds maxCredBlobLength, or extension is not supported for non-discoverable credentials), it returns the following CBOR map entry in the "extensions" field to to the authenticator:
    - "credBlob": false
- **authenticatorGetAssertion authenticator extension output**

- If the authenticator has the credBlob value for the credential, it returns the credBlob value in the following CBOR map entry in the "extensions" fields to the authenticator:
  - "credBlob": Byte String.
- If the authenticator does NOT have the credBlob value for the credential, it returns an empty Byte String in the following CBOR map entry in the "extensions" fields to the authenticator:
  - "credBlob": (empty) Byte String.

12.3. Large Blob Key (largeBlobKey)§

The credBlob extension allows for a small amount of opaque data to be stored with a credential. In contrast, this extension allows for a much larger amount of data to be stored in the large-blob array, protected by a key that is stored and accessed using this extension. Details of the interaction with the large-blob array are given in § 6.10.3 Large, per-credential blobs. This extension is mutually exclusive with the largeBlob extension.

Conceptually this extension extends the state of a discoverable credential with 32 bytes of opaque storage that may, or may not, be present for any given credential. This is called the **largeBlobKey**. Since this value is a random key, an authenticator MAY derive it as needed from other key material, rather than storing the value itself. If an authenticator does this, the same value MUST NOT be plausibly derivable via other means. For example, it MUST NOT also be obtainable via the hmac-secret extension using any salt that is predictable or constant across different credentials.

> NOTE: Client platforms SHOULD use the largeBlobKey registration extension when creating the credential if they wish to later use the largeBlobKey authentication extension to fetch the largeBlobKey. Authenticators MAY optionally generate a largeBlobKey for a credential if the Large Blob Key (largeBlobKey) extension is absent, but MUST NOT return an unsolicited largeBlobKey extension response or `largeBlobKey` (0x05) in the authenticatorMakeCredential response structure.

Platforms can detect support for this extension by checking for *all* of the following in the authenticatorGetInfo response:

1. `largeBlobKey` in the `extensions` field.
2. `largeBlobs` mapped to `true` in the `options` field.

Client extension input / output / processing
None. This extension is used to enable storage of large blobs in the large-blob array, which requires additional platform behaviour. It is not suitable to be directly exposed to RPs.

Authenticator input for authenticatorMakeCredential
"largeBlobKey": boolean.

Authenticator processing for authenticatorMakeCredential:

1. If the value of `largeBlobKey` is not `true`, return CTAP2_ERR_INVALID_OPTION. (The extension should be omitted rather than asserted to be false.)
2. If the `options` field of the authenticatorMakeCredential request does not map `rk` to `true`, return CTAP2_ERR_INVALID_OPTION.
3. If other processing steps for authenticatorMakeCredential complete successfully then update the new credential's state to store a freshly generated 32-byte key as its largeBlobKey.
4. Set the value of `largeBlobKey` (0x05) in the authenticatorMakeCredential response structure (i.e., *not* in the extensions field of the authenticator data) to the value of the generated largeBlobKey.

Authenticator authenticatorMakeCredential extension output
None. Since platforms cannot filter the content of the authenticator extension output, none is provided to avoid internal details of large-blob support leaking out of the abstraction layer.

Authenticator authenticatorGetAssertion extension input
"largeBlobKey": boolean

Authenticator authenticatorGetAssertion extension processing

1. If the value of `largeBlobKey` is not `true`, return CTAP2_ERR_INVALID_OPTION. (The extension should be omitted rather than asserted to be false.)
2. If other processing steps for authenticatorGetAssertion complete successfully, and the credential has an associated largeBlobKey, then set the value of `largeBlobKey` (0x07) in the authenticatorGetAssertion response structure (i.e., *not* in the extensions field of the authenticator data) to the stored `largeBlobKey`.

Authenticator authenticatorGetAssertion extension output
None. Since platforms cannot filter the content of the authenticator extension output, none is provided to avoid internal details of large-blob support leaking out of the abstraction layer.

12.4. Large Blob (largeBlob)§

This extension is an alternative to the to authenticatorLargeBlobs command and the largeBlobKey extension for authenticators that can accept the full contents of a largeBlob in an authenticatorGetAssertion message. Authenticators MUST NOT support both extensions.

The `largeBlob` extension closely mirrors the stucture of the WebAuthn extension of the same name. The major difference is that blob data is compressed in the CTAP version and the uncompressed size is stored with it.

Outputs are put into unsigned extension outputs so that the RP-observable behaviour is identical between the two styles of large blob support.

Authenticator input for authenticatorMakeCredential
"largeBlob": CBOR map matching the following CDDL:

```
largeblob-makeCredential-inputs = {
  support: "required" / "preferred"
}
```

Authenticator processing for authenticatorMakeCredential:

1. If the input does not conform to the given CDDL, return CTAP2_ERR_INVALID_CBOR.
2. If the authenticator can support large blobs in the newly created credential:
    1. Add an element to the unsigned extension outputs for this extension where the value is `{"supported": true}`.
3. Else:
    1. If `support` is “required” then return CTAP2_ERR_LARGE_BLOB_STORAGE_FULL.

(Authenticators MAY choose to always create new credentials with large blob capability, whether requested or not. However they MUST NOT return unsolicited output.)

Authenticator authenticatorMakeCredential extension output
None, as the output is in the unsigned extension output.

Authenticator authenticatorGetAssertion extension input
"largeBlob": CBOR map matching the following CDDL:

```
largeblob-inputs = {
  ? read : true
  ? write : bstr
  ? originalSize : uint
}
```

Authenticator authenticatorGetAssertion extension processing

1. If the input does not conform to the given CDDL, return CTAP2_ERR_INVALID_CBOR.
2. If the input contains the `read` member and neither of `write` nor `originalSize` members, or contains the `write` and `originalSize` members but not the `read` member, then continue. Otherwise return CTAP2_ERR_INVALID_CBOR.
3. If the `read` member is present:
    1. Fetch any largeBlob data for selected credentials. If there is none then stop processing this extension.
    2. Add an element to the unsigned extension outputs for this extension that conforms to `largeblob-outputs`, below, and which contains the compressed blob and its original size, as provided when it was written.
4. Else:
    1. Let the variable `written` be false.
    2. If the authenticatorGetAssertion request included a non-empty allowList, and the selected credential can store the large blob data, then save the contents of the `write` and `originalSize` inputs in the selected credential and set `written` to true.
    3. Add an element to the unsigned extension outputs for this extension that conforms to `largeblob-outputs`, below, and which contains a `written` member equal to the value of the `written` variable.

```
largeblob-outputs = {
  ? written : bool
  ? blob : bstr
  ? originalSize : uint
}
```

Authenticator authenticatorGetAssertion extension output
None, as the output is in the unsigned extension output.

12.5. Minimum PIN Length Extension (minPinLength)§

Extension identifier
`minPinLength`

This extension returns the current minimum PIN length value. This value does not decrease unless the authenticator is reset, in which case, all the credentials are reset. This extension is only applicable during credential creation.

See also § 7.4 Set Minimum PIN Length for the overall feature description.

NOTE: An example use case for this extension is: an organization supplies configured authenticators to their users, with a current minimum PIN length value tailored to the organization's requirements. Upon users registering their credentials with the organization's systems using the authenticators, the organization may use this extension to determine whether the current minimum PIN length continues to meet the organization's requirements.

Client extension input

`create()` : A boolean value to indicate that this extension is requested by the Relying Party.

```
partial dictionary AuthenticationExtensionsClientInputs {
  boolean minPinLength;
};
```

`get()` : Not applicable.

Client extension processing

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

Client extension output

None. Authenticator returns the result in authenticator extension output.

Authenticator extension input

Boolean asking for minimum PIN length value in Unicode code points. The platform sends the authenticatorMakeCredential request with the following CBOR map entry in the "extensions" field to the authenticator:

- "minPinLength": true

Authenticator extension processing

The authenticator checks whether the authenticatorMakeCredential's `rp.id` parameter is present on its `minPinLengthRPIDs` list. If so, the RP is authorized to receive the current minimum PIN length value. If not, the RP is not authorized to receive the current minimum PIN length value.

Authenticator extension output

- If the RP is

  ↪ **authorized, the authenticator sets the `minPinLength` return value to the current minimum PIN length value.**
  ↪ **not authorized, the authenticator ignores the extension and does not return any authenticator extension output.**

```
CDDL:
"minPinLength": uint
```

12.6. PIN Complexity Extension (pinComplexityPolicy)§

Extension identifier

`pinComplexityPolicy`

This extension returns the current PIN complexity policy value. This value does not change from TRUE to FALSE unless the authenticator is reset, in which case, all the credentials are reset. This extension is only applicable during credential creation.

See also § 7.5 Set PIN Complexity Policy for the overall feature description.

NOTE: An example use case for this extension is: an organization supplies configured authenticators to their users, with the current PIN complexity policy value tailored to the organization's requirements. Upon users registering their credentials with the organization's systems using the authenticators, the organization may use this extension to determine whether the current PIN complexity policy continues to meet the organization's requirements.

Client extension input

`create()` : A boolean value to indicate that this extension is requested by the Relying Party.

```
partial dictionary AuthenticationExtensionsClientInputs {
  boolean pinComplexityPolicy;
};
```

`get()` : Not applicable.

Client extension processing

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

Client extension output

None. The authenticator returns the result in the authenticator extension output.

Authenticator extension input

Boolean asking if there is a current PIN complexity policy configured. The platform sends the authenticatorMakeCredential request with the following CBOR map entry in the "extensions" field to the authenticator:

- "pinComplexityPolicy": true

Authenticator extension processing

The authenticator checks whether the authenticatorMakeCredential's `rp.id` parameter is present on its `minPinLengthRPIDs` list. If so, the RP is authorized to receive the current PIN complexity policy value. If not, the RP is not authorized to receive the current PIN complexity policy value.

Authenticator extension output

- If the RP is

  ↪ **authorized, the authenticator sets the `pinComplexityPolicy` return value to the current minimum PIN length value.**
  ↪ **not authorized, the authenticator ignores the extension and does not return any authenticator extension output.**

```
CDDL:
"pinComplexityPolicy": boolean
```

12.7. HMAC Secret Extension (hmac-secret)§

Extension identifier

`hmac-secret`

This extension is used by the platform to retrieve a symmetric secret from the authenticator when it needs to encrypt or decrypt data using that symmetric secret. This symmetric secret is scoped to a credential. The authenticator and the platform each only have the part of the complete secret to prevent offline attacks. This extension can be used to maintain different secrets on different machines. If authenticator supports this extension, authenticator MUST support it for both discoverable and non-discoverable credentials.

Client extension input

`create()` : A boolean value to indicate that this extension is requested by the Relying Party.

```
partial dictionary AuthenticationExtensionsClientInputs {
  boolean hmacCreateSecret;
};
```

`get()` : A JavaScript object defined as follows:

```
dictionary HMACGetSecretInput {
  required ArrayBuffer salt1;   // 32-byte random data
  ArrayBuffer salt2;  // Optional additional 32-byte random data
};

partial dictionary AuthenticationExtensionsClientInputs {
  HMACGetSecretInput hmacGetSecret;
};
```

The salt2 input is OPTIONAL. It can be used when the platform wants to roll over the symmetric secret in one operation.

Client extension processing

1. If present in a `create()`:
   1. If set to true, pass a CBOR true value as the authenticator extension input.
   2. If set to false, do not process this extension.
2. If present in a `get()`:
   1. Verify that salt1 is a 32-byte ArrayBuffer.
   2. If salt2 is present, verify that it is a 32-byte ArrayBuffer.
   3. Pass salt1 and, if present, salt2 as the authenticator extension input.

Client extension output

`create()`: Boolean true value indicating that the authenticator has processed the extension.

```
partial dictionary AuthenticationExtensionsClientOutputs {
  boolean hmacCreateSecret;
};
```

`get()`: A dictionary with the following data:

```
dictionary HMACGetSecretOutput {
  required ArrayBuffer output1;
  ArrayBuffer output2;
};

partial dictionary AuthenticationExtensionsClientOutputs {
  HMACGetSecretOutput hmacGetSecret;
};
```

Authenticator extension input

Same as the client extension input, except represented in CBOR.

Authenticator extension processing

- **authenticatorGetInfo additional behaviors**

  The authenticator indicates to the platform that it supports the "hmac-secret" extension via the "extensions" parameter in the authenticatorGetInfo response.

  > EXAMPLE 10
  > Sample CTAP2 authenticatorGetInfo response (CBOR):
  >
  > ```
  > {
  >   1: ["FIDO_2_0"],
  >   2: ["hmac-secret"],
  >   ...
  > }
  > ```

- **authenticatorMakeCredential additional behaviors**

  The platform sends the authenticatorMakeCredential request with the following CBOR map entry in the "extensions" field to the authenticator:

  - "hmac-secret": true

  > EXAMPLE 11
  > Sample CTAP2 authenticatorMakeCredential Request (CBOR):
  >
  > ```
  > {
  >   1: h'687134968222EC17202E42505F8ED2B16AE22F16BB05B88C25DB9E602645F141',
  >   ...
  >   6: {"hmac-secret": true},
  > }
  > ```

  - The authenticator generates two random 32-byte values (called `CredRandomWithUV` and `CredRandomWithoutUV`) and associates them with the credential.

    > NOTE: Authenticator SHOULD generate `CredRandomWithUV`/`CredRandomWithoutUV` and associate them with the credential, even if `hmac-secret` extension is not present in authenticatorMakeCredential request.

  - If the platform has sent the `hmac-secret` extension to the authenticator, then
    - If the authenticator succeeded in above step of generating `CredRandomWithUV`/`CredRandomWithoutUV` and associating it with the credential, it responds with the following CBOR map entry in the "extensions" fields to the platform:
      - `"hmac-secret": true`
    - Else (The authenticator did not succeed in above step of generating `CredRandomWithUV`/`CredRandomWithoutUV` and associating it with the credential), it responds with the following CBOR map entry in the "extensions" fields to the platform:
      - `"hmac-secret": false`
  - Else (the platform has not sent the hmac-secret extension to the authenticator)
    - Authenticator does not add any response from this extension to the "extensions" field of the authenticatorMakeCredential response.

- **authenticatorGetAssertion additional behaviors**
  - The platform gets sharedSecret from the authenticator.
  - The platform sends the authenticatorGetAssertion request with the following CBOR map entry in the "extensions" field to the authenticator:
    - "hmac-secret":
      - keyAgreement(0x01): public key of platform key-agreement key.
      - saltEnc(0x02): Encryption of the one or two salts (called salt1 (32 bytes) and salt2 (32 bytes)) using the shared secret as follows:
        - One salt case: encrypt(shared secret, salt1)
        - Two salt case: encrypt(shared secret, salt1 || salt2)
      - saltAuth(0x03): authenticate(shared secret, saltEnc)
      - pinUvAuthProtocol(0x04): (optional) as selected when getting the shared secret. CTAP2.1 platforms MUST include this parameter if the value of `pinUvAuthProtocol` is not 1.

EXAMPLE 12

Sample CTAP2 authenticatorGetAssertion Request (CBOR):

```
{
  1: "example.com",
  2: h'687134968222EC17202E42505F8ED2B16AE22F16BB05B88C25DB9E602645F141',
  ...
  4: {
    "hmac-secret":
      {
        1:
          {
            1: 2,
            3: -25,
            -1: 1,
            -2: h'0DE6479775C5B704BF780073809DE1B36A29132E187709C1E364F299F8847769'
,
            -3: h'3BBE9BEDCC1AC8328BA6397A5F46AF85FC7C51B35BEDFD9E3E47AC6F34248B35
'
          },
        2: h'59E195FC58C614C07C99F587495F374871E9873AD37D5BCA1EED200926C3C6BA528D77
48AF9592BD7E7A88051887F214E13CFDF406C3A1C57D529BABF987D4A',
        3: h'17B93F3BDB95380ED512EC6F542CE140'
      }
   }
}
```

- The authenticator performs the following operations when processing this extension:
  - If pinUvAuthProtocol is absent and a pinUvAuthProtocol value of1 is supported by the authenticator, let the value of pinUvAuthProtocol be `1`
  - If pinUvAuthProtocol is absent and a pinUvAuthProtocol value of1 is not supported by the authenticator, then return CTAP2_ERR_PIN_AUTH_INVALID.
  - If "up" is set to false, authenticator returns CTAP2_ERR_UNSUPPORTED_OPTION.
  - The authenticator waits for user consent.
  - If request asks for user verification, authenticator waits for user verification.
    - If user verification is requested via Client PIN mechanism, verify the user by verifying the Client PIN parameters in the request as mentioned in the authenticatorGetAssertion steps.
    - If user verification is requested via a built-in user verification method, verify the user by built-in user verification method as mentioned in the authenticatorGetAssertion steps.
  - The authenticator calls decapsulate on the provided platform key-agreement key to obtain a shared secret.
  - Authenticator calls verify(shared secret, saltEnc, saltAuth)
    - If the verification fails, return CTAP2_ERR_PIN_AUTH_INVALID.
  - Authenticator obtains salt1 and salt2 by calling decrypt(shared secret, saltEnc). If the decryption fails, or if the result is not 32 or 64 bytes long, return CTAP1_ERR_INVALID_PARAMETER. Otherwise salt1 is the first 32 bytes of the result and salt2 is the remaining bytes, if any.
  - The authenticator chooses which `CredRandom` to use for next step based on whether user verification was done or not in above steps.
    - If uv bit is set to 1 in the response, let`CredRandom` be `CredRandomWithUV`.
    - If uv bit is set to 0 in the response, let`CredRandom` be `CredRandomWithoutUV`.
  - If the authenticator cannot find corresponding CredRandom associated with the credential, authenticator ignores this extension and does not add any response from this extension to "extensions" field of the authenticatorGetAssertion response.
  - The authenticator generates one or two HMAC-SHA-256 values, depending upon whether it received one salt (32 bytes) or two salts (64 bytes):
    - output1: `HMAC-SHA-256(CredRandom, salt1)`
    - output2: `HMAC-SHA-256(CredRandom, salt2)`
  - The authenticator returns output1 and (when there were two salts) output2, encrypted to the platform using the shared secret, as part of "extensions" parameter:
    - One salt case: `"hmac-secret"`: encrypt(shared secret, output1)
    - Two salt case: `"hmac-secret"`: encrypt(shared secret, output1 || output2)

EXAMPLE 13

Sample "extensions" field value in the authenticatorData:

```
{ "hmac-secret": h'1F91526CAE456E4CBB71C4DDE7BB877157E6E54DFED3015D7D4DBB2269AFCDE6A91E
D267EBBF848EB95A68E79C7AC705E351D543DB0165887D6290FD47A40C4' }
```

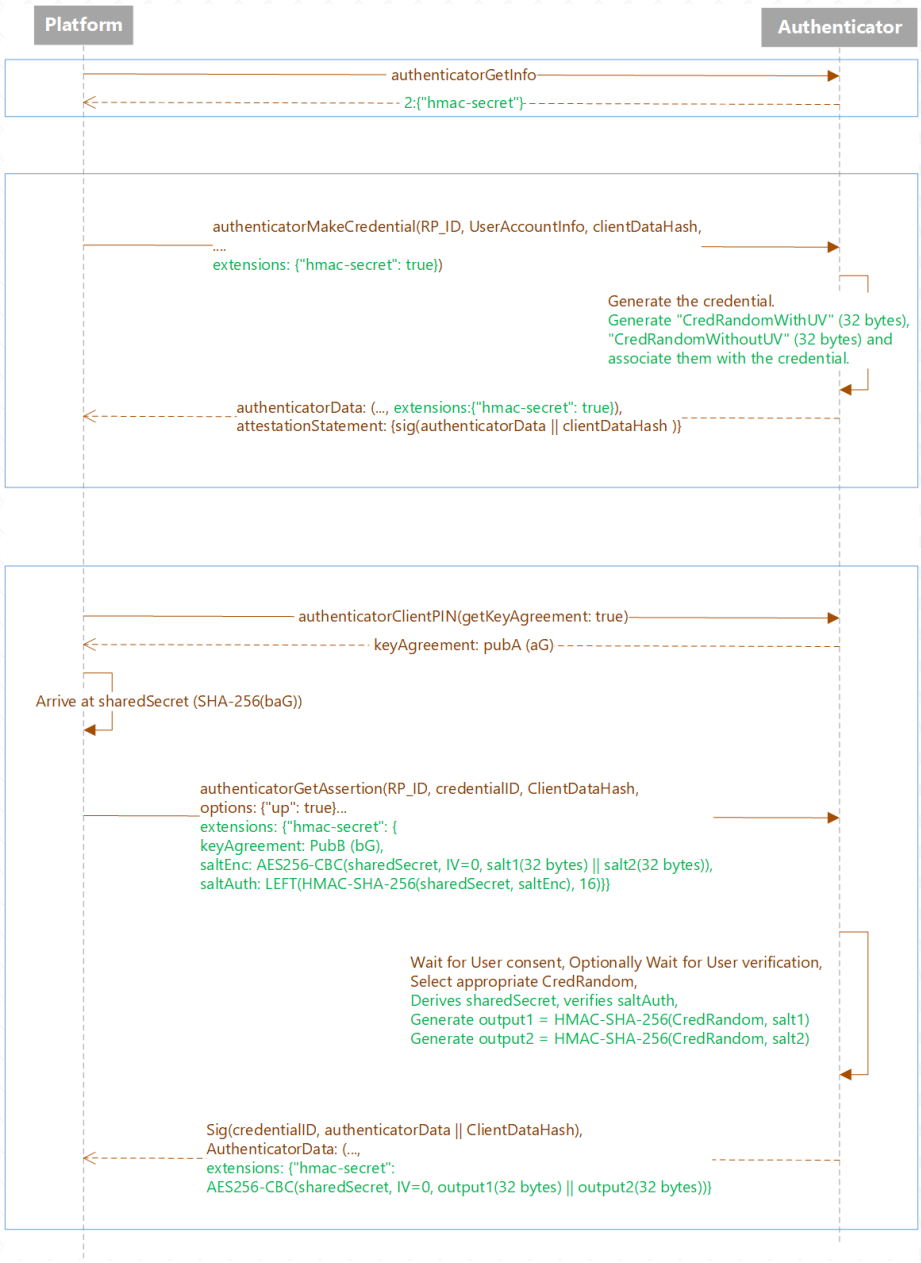

***Figure 6** hmac-secret*

Authenticator extension output

Same as the client extension output, except represented in CBOR.

12.8. HMAC Secret MakeCredential Extension (hmac-secret-mc)

Extension identifier

`hmac-secret-mc`

This extension is similar to the above hmac-secret extension where a symmetric secret can be obtained when creating a key. If the authenticator supports this extension, the hmac-secret extension MUST be supported as well, and the authenticator MUST support it for both discoverable and non-discoverable credentials. This extension is only applicable for authenticatorMakeCredential, and the hmac-secret extension MUST also be present with the value of "hmac-secret" set to true. The authenticator MUST return CTAP2_ERR_MISSING_PARAMETER when they receive this extension without the "hmac-secret" extension.

Client extension input

- **Not Applicable**

Client extension output

- **Not Applicable**

Authenticator extension input

- **authenticatorMakeCredential additional behaviors**
  - This extension input is the same as the hmac secret extension's getAssertion input

Authenticator extension processing

- **authenticatorMakeCredential additional behaviors**
  - This extension processing is the same as the hmac secret extension's getAssertion processing

Authenticator extension output

Same as the hmac secret extension's getAssertion output.

12.9. Third-Party Payment authentication (thirdPartyPayment)§

This extension allows a Relying Party to indicate that a credential can be used for Payment authentication initiated by a party (website or native application) that is not the Relying Party. The platform is responsible for determining what constitutes a **Payment authentication** - the W3C [secure-payment-confirmation] specification is one example that a platform may implement.

A credential marked this way is referred to as **third-party payment enabled**, and the authenticator stores this information for future retrieval. If the authenticator supports this extension, the authenticator MUST support it for both discoverable and non-discoverable credentials.

Extension identifier

`thirdPartyPayment`

Client extension input / output / processing None. The client processing steps are platform-dependent, e.g. see [secure-payment-confirmation] for the web platform.
Authenticator extension input

- **authenticatorMakeCredential authenticator extension input**
  - The platform sends the authenticatorMakeCredential request with the following CBOR map entry in the "extensions" field to the authenticator:
    - "thirdPartyPayment": true
- **authenticatorGetAssertion authenticator extension input**
  - The platform sends the authenticatorGetAssertion request with the following CBOR map entry in the "extensions" field to the authenticator:
    - "thirdPartyPayment" : true

Authenticator extension processing

The `thirdPartyPayment` boolean is persisted with the Credential during authenticatorMakeCredential and returned during authenticatorGetAssertion.

Authenticator extension output

- **authenticatorMakeCredential authenticator extension output** None.
- **authenticatorGetAssertion authenticator extension output**
  - If the credential was created with the `thirdPartyPayment` extension specified, the authenticator returns the following CBOR map entry in the "extensions" fields to the platform:
    - "thirdPartyPayment": true
  - Otherwise the authenticator returns the following CBOR map entry in the "extensions" fields to the platform:
    - "thirdPartyPayment": false

13. Related Documents§

The following documents are published by other organisations and are not referenced by this specification but may be relevant to the same audience. They are gathered here purely as informational resources and are not necessarily endorsed by FIDO.

1. Windows provides a WebAuthn-like API to applications.
2. Android provides a WebAuthn-like API and provides a mechanism for apps to claim domain names as valid RP IDs.

14. IANA Considerations§

14.1. WebAuthn Extension Identifier Registrations§

This section registers the extension identifier values defined in Section § 12 Defined Extensions in the IANA "WebAuthn Extension Identifiers" registry [IANA-WebAuthn-Registries] established by [RFC8809].

- WebAuthn Extension Identifier: credProtect
- This registration extension allows relying parties to specify a credential protection policy when creating a credential. Additionally, authenticators may choose to establish a default credential protection policy greater than `userVerificationOptional` (the lowest level) and unilaterally enforce such policy.
- Specification Document: Section § 12.1 Credential Protection (credProtect) of this specification

- WebAuthn Extension Identifier: credBlob
- Description: This registration extension and authentication extension enables RPs to provide a small amount of extra credential configuration information (the credBlob value) to the authenticator when a credential is made.
- Specification Document: Section § 12.2 Credential Blob (credBlob) of this specification

- WebAuthn Extension Identifier: largeBlobKey
- Description: This client platform-only extension provides for storage and retrieval of a per-credential key that is used by the client platform when writing and reading elements in the large-blob array.
- Specification Document: Section § 12.3 Large Blob Key (largeBlobKey) of this specification

- WebAuthn Extension Identifier: minPinLength
- Description: This registration extension returns the current minimum PIN length value to the Relying Party.
- Specification Document: Section § 12.5 Minimum PIN Length Extension (minPinLength) of this specification

- WebAuthn Extension Identifier: hmac-secret
- Description: This registration extension and authentication extension enables the platform to retrieve a symmetric secret scoped to the credential from the authenticator.
- Specification Document: Section § 12.7 HMAC Secret Extension (hmac-secret) of this specification

15. Security Considerations§

See FIDO Security Reference document [FIDOSecRef].

Index§

Terms defined by this specification§

third-party payment enabled

transports

transportsForReset

tunnel service

uint32LittleEndian

uint64LittleEndian

uint8

up

- dfn for getAssert
- dfn for makeCred

usage timer

User action timeout

user consent

user presence

userPresent flag

user present time limit

userVerificationMgmtPreview

userVerificationOptional

userVerificationOptionalWithCredentialIDList

userVerificationRequired

userVerified flag

uv

- dfn for getAssert
- dfn for getInfo
- dfn for makeCred

uvAcfg

uvBioEnroll

uvCountSinceLastPinEntry

uvRetries

vendorCommandId

Vendor-facilitated enterprise attestation

vendorPrototypeConfigCommands

verify

versions

Terms defined by reference§

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

- create()
- get()

[DIGITAL-CREDENTIALS] defines the following terms:

- digital credentials api

[JSON-SCHEMA] defines the following terms:

- json schema

[WebAuthn-2] defines the following terms:

- assertion signature
- attestation
- attestation object
- attestation statement format identifier
- attested credential data
- authenticator
- authenticator data
- authenticator extension input
- authenticator extension output
- authenticatorgetassertion operation
- authenticatormakecredential operation
- client platform
- client side
- credential key pair
- discoverable credential
- enterprise attestation

extension identifier
generating an attestation object
hash of the serialized client data
lookup credential source by credential id algorithm
private key
public key credential
public key credential source
relying party identifier
rp id
server-side credential
user handle
user verification

[WEBAUTHN-3] defines the following terms:

AuthenticationExtensionsClientInputs
AuthenticationExtensionsClientOutputs
PublicKeyCredentialDescriptor
PublicKeyCredentialParameters
PublicKeyCredentialRpEntity
PublicKeyCredentialUserEntity
authenticatorSelection
displayName
id
largeblob extension
name
type
unsigned extension output
userVerification (for AuthenticatorSelectionCriteria)
userVerification (for PublicKeyCredentialRequestOptions)

[WEBIDL] defines the following terms:

ArrayBuffer
USVString
boolean

References§

Normative References§

[BTASSNUM]
Bluetooth Assigned Numbers. URL: https://www.bluetooth.org/en-us/specification/assigned-numbers

[BTCCC]
Client Characteristic Configuration. Bluetooth Core Specification 4.0, Volume 3, Part G, Section 3.3.3.3 URL: https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=229737

[BTCORE]
Bluetooth Core Specification 4.0. URL: https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=229737

[BTDIS]
Device Information Service v1.1. URL: https://www.bluetooth.com/specifications/adopted-specifications

[BTGAP]
Generic Access Profile. Bluetooth Core Specification 4.0, Volume 3, Part C, Section 12 URL: https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=229737

[BTGAS]
Generic Access Profile service. Bluetooth Core Specification 4.0, Volume 3, Part C, Section 12 URL: https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=229737

[BTPESTK]
Passkey Entry. Bluetooth Core Specification 4.0, Volume 3, Part H, Section 2.3.5.3 URL: https://www.bluetooth.com/specifications/adopted-specifications

[BTSD]
Bluetooth Service Data AD Type. Bluetooth Core Specification 4.0, Volume 3, Part C, Section 11 URL: https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=229737

[BTXPLAD]
Bluetooth TX Power AD Type. Bluetooth Core Specification 4.0, Volume 3, Part C, Section 11 URL: https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=229737

[CC1V3-1R5]
CCMB-2017-04-001 Common Criteria for Information Technology Security Evaluation, Part 1: Introduction and general model. April 2017. URL: https://www.commoncriteriaportal.org/files/ccfiles/CCPART1V3.1R5.pdf

[CMVP]
Implementation Guidance for FIPS 140-2 and the Cryptographic Module Validation Program - CMVP December 3, 2019. URL: https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Module-Validation-Program/documents/fips140-2/FIPS1402IG.pdf

[CommonCriteria]
CCRA Members. *Common Criteria Publications*. Work in Progress. URL: http://www.commoncriteriaportal.org/cc/

[CREDENTIAL-MANAGEMENT-1]
Nina Satragno; Marcos Caceres. *Credential Management Level 1*. URL: https://w3c.github.io/webappsec-credential-management/

[CSPN]
CSPN certification, Produits, Formulaires et Méthodologies URL: https://www.ssi.gouv.fr/administration/produits-certifies/cspn/les-procedures-formulaires-et-methodologies/

[DIGITAL-CREDENTIALS]
Digital Credentials. Draft Community Group Report. URL:https://wicg.github.io/digital-credentials/

[FIDOAuthenticatorSecurityRequirements]
Rolf Lindemann; Dr. Joshua E. Hill; Douglas Biggs. *FIDO Authenticator Security Requirements*. November 2020. Final Draft. URL: https://fidoalliance.org/specs/fido-security-requirements/fido-authenticator-security-requirements-v1.4-fd-20201102.html

[FIDORegistry]
R. Lindemann; et al. *FIDO Registry of Predefined Values*. 23 May 2022. Proposed Standard. URL: https://fidoalliance.org/specs/common-specs/fido-registry-v2.2-ps-20220523.html

[FIDOSecRef]
R. Lindemann; et al. *FIDO Security Reference*. 23 May 2022. Proposed Standard. URL: https://fidoalliance.org/specs/common-specs/fido-security-ref-v2.1-ps-20220523.html

[FIPS140-2]
FIPS PUB 140-2: Security Requirements for Cryptographic Modules. May 2001. URL: http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf

[FIPS140-3]
FIPS PUB 140-3: Security Requirements for Cryptographic Modules. March 2019. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.140-3.pdf

[IANA-COSE-ALGS-REG]
Jim Schaad; et al. *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/

[ISO7816-4]
ISO 7816-4: Identification cards - Integrated circuit cards; Part 4: Organization, security and commands for interchange. 2013-04. URL: https://www.iso.org/standard/54550.html

[JSON-SCHEMA]
Austin Wright; et al. *JSON Schema: A Media Type for Describing JSON Documents*. 10 June 2022. Internet-Draft. URL: https://datatracker.ietf.org/doc/html/draft-bhutton-json-schema

[RFC1951]
P. Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. May 1996. Informational. URL: https://www.rfc-editor.org/rfc/rfc1951

[RFC2397]
L. Masinter. *The "data" URL scheme*. August 1998. Proposed Standard. URL:https://www.rfc-editor.org/rfc/rfc2397

[RFC5116]
D. McGrew. *An Interface and Algorithms for Authenticated Encryption*. January 2008. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc5116

[RFC5869]
H. Krawczyk; P. Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. May 2010. Informational. URL: https://www.rfc-editor.org/rfc/rfc5869

[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. RFC. URL: https://www.rfc-editor.org/rfc/rfc8949.html

[RFC9052]
J. Schaad. *CBOR Object Signing and Encryption (COSE): Structures and Process*. August 2022. Internet Standard. URL: https://www.rfc-editor.org/rfc/rfc9052

[SEC1V2]
SEC1: Elliptic Curve Cryptography, Version 2.0 May 2009. URL:http://secg.org/download/aid-780/sec1-v2.pdf

[SECURE-PAYMENT-CONFIRMATION]
Rouslan Solomakhin (Google); Stephen McGruer (Google). *Secure Payment Confirmation*. 31 August 2021. TR. URL: https://www.w3.org/TR/secure-payment-confirmation/

[SP800-56A]
NIST Special Publication 800-56A: Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography (Revised). March 2007 URL: https://csrc.nist.gov/publications/nistpubs/800-56A/SP800-56A_Revision1_Mar08-2007.pdf

[U2FBIe]

D. Balfanz. *FIDO Bluetooth® Specification*. Proposed Standard. URL: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-bt-protocol-v1.2-ps-20170411.html

[U2FNfc]
D. Balfanz. *FIDO NFC Protocol Specification*. Proposed Standard. URL: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-nfc-protocol-v1.2-ps-20170411.html

[U2FRawMsgs]
D. Balfanz; J. Ehrensvard; J. Lang. *FIDO U2F Raw Message Formats v1.2* Proposed Standard. URL: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-raw-message-formats-v1.2-ps-20170411.html

[U2FUsbHid]
D. Balfanz. *FIDO U2F HID Protocol Specification*. Proposed Standard. URL: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-hid-protocol-v1.2-ps-20170411.html

[WebAuthn]
Dirk Balfanz (Google); et al. *Web Authentication: An API for accessing Public Key Credentials Level 2* 8 April 2021. TR. URL: https://www.w3.org/TR/webauthn-2/

[WebAuthn-2]
Jeff Hodges; et al. *Web Authentication: An API for accessing Public Key Credentials - Level 2* URL: https://w3c.github.io/webauthn/

[WEBAUTHN-3]
Michael Jones; Akshay Kumar; Emil Lundberg. *Web Authentication: An API for accessing Public Key Credentials - Level 3*. URL: https://w3c.github.io/webauthn/

[WEBIDL]
Edgar Chen; Timothy Gu. *Web IDL Standard*. Living Standard. URL: https://webidl.spec.whatwg.org/

Informative References§

[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

[RFC6090]
D. McGrew; K. Igoe; M. Salter. *Fundamental Elliptic Curve Cryptography Algorithms*. February 2011. Informational. URL: https://www.rfc-editor.org/rfc/rfc6090

[RFC8701]
D. Benjamin. *Applying Generate Random Extensions And Sustain Extensibility (GREASE) to TLS Extensibility*. January 2020. Informational. URL: https://www.rfc-editor.org/rfc/rfc8701

IDL Index§

```
partial dictionary AuthenticationExtensionsClientInputs {
  USVString credentialProtectionPolicy;
  boolean enforceCredentialProtectionPolicy = false;
};

partial dictionary AuthenticationExtensionsClientInputs {
  ArrayBuffer credBlob;
};

partial dictionary AuthenticationExtensionsClientInputs {
  boolean getCredBlob;
};

partial dictionary AuthenticationExtensionsClientOutputs {
  boolean credBlob;
};

partial dictionary AuthenticationExtensionsClientOutputs {
  ArrayBuffer getCredBlob;
};

partial dictionary AuthenticationExtensionsClientInputs {
  boolean minPinLength;
};

partial dictionary AuthenticationExtensionsClientInputs {
  boolean pinComplexityPolicy;
};

partial dictionary AuthenticationExtensionsClientInputs {
  boolean hmacCreateSecret;
};

dictionary HMACGetSecretInput {
  required ArrayBuffer salt1;   // 32-byte random data
  ArrayBuffer salt2;  // Optional additional 32-byte random data
};

partial dictionary AuthenticationExtensionsClientInputs {
  HMACGetSecretInput hmacGetSecret;
};

partial dictionary AuthenticationExtensionsClientOutputs {
  boolean hmacCreateSecret;
};

dictionary HMACGetSecretOutput {
  required ArrayBuffer output1;
  ArrayBuffer output2;
};

partial dictionary AuthenticationExtensionsClientOutputs {
  HMACGetSecretOutput hmacGetSecret;
};
```