Client to Authenticator Protocol (CTAP)

Proposed Standard, January 30, 2019

This version:
https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html

Previous Versions:
https://fidoalliance.org/specs/fido-v2.0-id-20180227/

Issue Tracking:
GitHub

Editors:
Christiaan Brand (Google)
Alexei Czeskis (Google)
Jakob Ehrensvärd (Yubico)
Michael B. Jones (Microsoft)
Akshay Kumar (Microsoft)
Rolf Lindemann (Nok Nok Labs)
Adam Powers (FIDO Alliance)
Johan Verrept (OneSpan)

Former Editors:
Matthieu Antoine (Gemalto)
Arnar Birgisson (Google)
Vijay Bharadwaj (Microsoft)
Mirko J. Ploch (SurePassID)

Contributors:
Jeff Hodges (PayPal)

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.

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 is expected to have a mechanism to obtain a user gesture. Possible examples of user gestures include: as a consent button, password, a PIN, a biometric or a combination of these.

Prior to executing this protocol, the client/platform (referred to as*host* hereafter) and roaming authenticator (referred to as *authenticator* hereafter) must establish a confidential and mutually authenticated data transport channel. This specification does not specify the details of how such a channel is established, nor how transport layer security must be achieved.

1.1. Relationship to Other Specifications§

This specification is part of the FIDO2 project which includes this CTAP and the[FIDOServerGuidelines] specifications, 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 theCTAP2 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).

The [U2FUsbHid], [U2FNfc], [U2FBle], and [U2FRawMsgs] specifications, specifically, are superseded by this specification.

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.

Using CTAP2 with CTAP1/U2F authenticators is defined inInteroperating with CTAP1/U2F authenticators.

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].

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 platform and an authenticator is as follows:

1. Platform establishes the connection with the authenticator.
2. Platform gets information about the authenticator using authenticatorGetInfo command, which helps it determine the capabilities of the authenticator.
3. Platform sends a command for an operation if the authenticator is capable of supporting it.
4. Authenticator replies with response data or error.

5. 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.

The authenticator API has the following methods and data structures.

5.1. 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, which explicitly correspond to those defined in The authenticatorMakeCredential

operation section of the Web Authentication specification:

Parameter name	Data type	Required?	Definition
clientDataHash (0x01)	Byte Array	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 of type text string, (optionally) a human-friendly RP name of type text string, and (optionally) a URL of type text string, referencing a RP icon image. 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/icon for single-factor scenarios.
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 array, (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). The authenticator associates the created public key credential with the account identifier, and MAY also associate any or all of the user name, user display name, and image data (pointed to by the URL, if any). The user name,

			display name, and URL 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, display name and URL 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.
pubKeyCredParams (0x04)	CBOR Array	Required	A sequence of CBOR maps consisting of pairs of PublicKeyCredentialType (a string) and cryptographic algorithm (a positive or negative integer), where algorithm identifiers are values that SHOULD be registered in the IANA COSE Algorithms registry [IANA-COSE-ALGS-REG]. This sequence is ordered from most preferred (by the RP) to least preferred.
excludeList (0x05)	Sequence of PublicKeyCredentialDescriptors	Optional	A sequence of PublicKeyCredentialDescriptor structures, as specified in [WebAuthn]. The authenticator returns an error if the authenticator already contains one of the credentials enumerated in this sequence. This allows RPs to limit the creation of multiple credentials for the same account on a single authenticator.
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.
pinAuth (0x08)	Byte Array	Optional	First 16 bytes of HMAC-SHA-256 of clientDataHash using pinToken which platform got from the authenticator: `HMAC-SHA-256(pinToken, clientDataHash)`.
pinProtocol (0x09)	Unsigned Integer	Optional	PIN protocol version chosen by the client

The following values are defined for use in the options parameter. All options are booleans.

Key	Default value	Definition
rk	false	resident key: Instructs the authenticator to store the key material on the device.
uv	false	user verification: Instructs the authenticator to require a gesture that verifies the user to complete the request. Examples of such gestures are fingerprint scan or a PIN.

Note that 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	Not present in the current version of CTAP. Authenticators are assumed to always check user presence.
requireUserVerification	options.uv or pinAuth/pinProtocol
credTypesAndPubKeyAlgs	pubKeyCredParams
excludeCredentialDescriptorList	excludeList
extensions	extensions

Note that 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 that a text string is a UTF-8 encoded string (CBOR major type 3).

When an `authenticatorMakeCredential` request is received, the authenticator performs the following procedure:

1. If the excludeList parameter is present and contains a credential ID that is present on this authenticator and bound to the specified rpId, wait for user presence, then terminate this procedure and return error code CTAP2_ERR_CREDENTIAL_EXCLUDED. User presence check is required for CTAP2 authenticators before the RP gets told that the token is already registered to behave similarly to CTAP1/U2F authenticators.
2. If the pubKeyCredParams parameter does not contain a valid COSEAlgorithmIdentifier value that is supported by the authenticator, terminate this procedure and return error code CTAP2_ERR_UNSUPPORTED_ALGORITHM.
3. If the options parameter is present, process all the options. If the option is known but not supported, terminate this procedure and return CTAP2_ERR_UNSUPPORTED_OPTION. If the option is known but not valid for this command, terminate this procedure and return CTAP2_ERR_INVALID_OPTION. Ignore any options that are not understood. Note that because this specification defines normative behaviors for them, all authenticators MUST understand the "rk", "up", and "uv" options.
4. Optionally, if the extensions parameter is present, process any extensions that this authenticator supports. Authenticator extension outputs generated by the authenticator extension processing are returned in the authenticator data.
5. If pinAuth parameter is present and pinProtocol is 1, verify it by matching it against first 16 bytes of HMAC-SHA-256 of clientDataHash parameter using pinToken: `HMAC- SHA-256(pinToken, clientDataHash)`.
    - If the verification succeeds, set the "uv" bit to 1 in the response.
    - If the verification fails, return CTAP2_ERR_PIN_AUTH_INVALID error.
6. If pinAuth parameter is not present and clientPin been set on the authenticator, return CTAP2_ERR_PIN_REQUIRED error.
7. If pinAuth parameter is present and the pinProtocol is not supported, return CTAP2_ERR_PIN_AUTH_INVALID.
8. If the authenticator has a display, show the items contained within the user and rp parameter structures to the user. Alternatively, request user interaction in an authenticator-specific way (e.g., flash the LED light). Request permission to create a credential. If the user declines permission, return the CTAP2_ERR_OPERATION_DENIED error.
9. Generate a new credential key pair for the algorithm specified.
10. If "rk" in options parameter is set to true:
    - If a credential for the same RP ID and account ID already exists on the authenticator, overwrite that credential.
    - Store the user parameter along the newly-created key pair.
    - If authenticator does not have enough internal storage to persist the new credential, return CTAP2_ERR_KEY_STORE_FULL.
11. Generate an attestation statement for the newly-created key using clientDataHash.

On success, the authenticator returns an attestation object in its response as defined in [WebAuthn]:

Member name	Data type	Required?	Definition
authData (0x01)	Byte Array	Required	The authenticator data object.
fmt (0x02)	String	Required	The attestation statement format identifier.
			The attestation statement, whose format is identified by the "fmt" object

			member. The client treats it as an opaque object.
attStmt (0x03)	Byte Array, the structure of which depends on the attestation statement format identifier	Required	

5.2. 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, which explicitly 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 Array	Required	Hash of the serialized client data collected by the host. See [WebAuthn].
allowList (0x03)	Sequence of PublicKeyCredentialDescriptors	Optional	A sequence of PublicKeyCredentialDescriptor structures, each denoting a credential, as specified in [WebAuthn]. If this parameter is present and has 1 or more entries, 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.
pinAuth (0x06)	Byte Array	Optional	First 16 bytes of HMAC-SHA-256 of clientDataHash using pinToken which platform got from the authenticator: `HMAC-SHA-256(pinToken, clientDataHash)`.
pinProtocol (0x07)	Unsigned Integer	Optional	PIN protocol version selected by client.

The following values are defined for use in the options parameter. All options are booleans.

Key	Default value	Definition
up	true	user presence: Instructs the authenticator to require user consent to complete the operation.
uv	false	user verification: Instructs the authenticator to require a gesture that verifies the user to complete the request. Examples of such gestures are fingerprint scan or a PIN.

Note that 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
requireUserVerification	options.uv or pinAuth/pinProtocol
extensions	extensions

When an `authenticatorGetAssertion` request is received, the authenticator performs the following procedure:

1. Locate all credentials that are eligible for retrieval under the specified criteria:
    - If an allowList is present and is non-empty, locate all denoted credentials present on this authenticator and bound to the specified rpId.
    - If an allowList is not present, locate all credentials that are present on this authenticator and bound to the specified rpId.
    - Let `numberOfCredentials` be the number of credentials found.
2. If pinAuth parameter is present and pinProtocol is 1, verify it by matching it against first 16 bytes of HMAC-SHA-256 of clientDataHash parameter using pinToken: `HMAC-SHA-256(pinToken, clientDataHash)`.
    - If the verification succeeds, set the "uv" bit to 1 in the response.
    - If the verification fails, return CTAP2_ERR_PIN_AUTH_INVALID error.
3. If pinAuth parameter is present and the pinProtocol is not supported, return CTAP2_ERR_PIN_AUTH_INVALID.
4. If pinAuth parameter is not present and clientPin has been set on the authenticator, set the "uv" bit to 0 in the response.
5. If the options parameter is present, process all the options. If the option is known but not supported, terminate this procedure and return CTAP2_ERR_UNSUPPORTED_OPTION. If the option is known but not valid for this command, terminate this procedure and return CTAP2_ERR_INVALID_OPTION. Ignore any options that are not understood. Note that because this specification defines normative behaviors for them, all authenticators MUST understand the "rk", "up", and "uv" options.
6. Optionally, if the extensions parameter is present, process any extensions that this authenticator supports. Authenticator extension outputs generated by the authenticator extension processing are returned in the authenticator data.
7. Collect user consent if required. This step MUST happen before the following steps due to privacy reasons (i.e., authenticator cannot disclose existence of a credential until the user interacted with the device):
    - If the "uv" option was specified and set to true:
        - If device doesn't support user-identifiable gestures, return the CTAP2_ERR_UNSUPPORTED_OPTION error.
        - Collect a user-identifiable gesture. If gesture validation fails, return the CTAP2_ERR_OPERATION_DENIED error.
    - If the "up" option was specified and set to true, collect the user's consent.
        - If no consent is obtained and a timeout occurs, return the CTAP2_ERR_OPERATION_DENIED error.
8. If no credentials were located in step 1, return CTAP2_ERR_NO_CREDENTIALS.
9. If more than one credential was located in step 1 and allowList is present and not empty, select any applicable credential and proceed to step 12. Otherwise, order the credentials by the time when they were created in reverse order. The first credential is the most recent credential that was created.

10. If authenticator does not have a display:
    - Remember the authenticatorGetAssertion parameters.
    - Create a credential counter(`credentialCounter`) and set it 1. This counter signifies how many credentials are sent to the platform by the authenticator.
    - Start a timer. This is used during authenticatorGetNextAssertion command. This step is optional if transport is done over NFC.
    - Update the response to include the first credential's publicKeyCredentialUserEntity information and numberOfCredentials. User identifiable information (name, DisplayName, icon) inside publicKeyCredentialUserEntity MUST not be returned if user verification is not done by the authenticator.
11. If authenticator has a display:
    - Display all these credentials to the user, using their friendly name along with other stored account information.
    - Also, display the rpId of the requester (specified in the request) and ask the user to select a credential.
    - 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.
12. Sign the clientDataHash along with authData with the selected credential, using the structure specified in [WebAuthn].

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

Member name	Data type	Required?	Definition
credential (0x01)	PublicKeyCredentialDescriptor	Optional	PublicKeyCredentialDescriptor structure containing the credential identifier whose private key was used to generate the assertion. May be omitted if the allowList has exactly one Credential.
authData (0x02)	Byte Array	Required	The signed-over contextual bindings made by the authenticator, as specified in [WebAuthn].
signature (0x03)	Byte Array	Required	The assertion signature produced by the authenticator, as specified in [WebAuthn].
			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

user (0x04)	PublicKeyCredentialUserEntity	Optional	information is not present for U2F credentials. FIDO Devices - server resident credentials: For server resident credentials on FIDO devices, this parameter is optional as server resident credentials behave same as U2F credentials where they are discovered given the user information on the RP. Authenticators optionally MAY store user information inside the credential ID. FIDO devices - device resident credentials: For device resident keys 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 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. This member is required when more than one account for the RP and the authenticator does not have a display. Omitted when returned for the authenticatorGetNextAssertion method.

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) 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 (a typical "smartcard scenario")
"up"=1 "uv"=1	User verification performed and physical user presence verified

5.3. 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.

This method takes no arguments as it is always follows a call to authenticatorGetAssertion or authenticatorGetNextAssertion.

When such a request 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.
4. Sign the clientDataHash along with authData with the credential using `credentialCounter` as index (e.g., credentials[n] assuming 0-based array), using the structure specified in [WebAuthn].
5. Reset the timer. This step is optional if transport is done over NFC.
6. Increment `credentialCounter`.

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

5.3.1. 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.

5.4. authenticatorGetInfo (0x04)§

Using this method, the host can request that the authenticator report a list of all supported protocol versions, supported extensions, AAGUID of the device, and its capabilities. This method takes no inputs.

On success, the authenticator returns:

Member name	Data type	Required?	Definition
versions (0x01)	Sequence of strings	Required	List of supported versions. Supported versions are: "FIDO_2_0" for CTAP2 / FIDO2 / Web Authentication authenticators and "U2F_V2" for CTAP1/U2F authenticators.
extensions (0x02)	Sequence of strings	Optional	List of supported extensions.
aaguid (0x03)	Byte String	Required	The claimed AAGUID. 16 bytes in length and encoded the same as MakeCredential AuthenticatorData, as specified in [WebAuthn].
options (0x04)	Map	Optional	List of supported options.
maxMsgSize (0x05)	Unsigned Integer	Optional	Maximum message size supported by the authenticator.
pinProtocols (0x06)	Array of Unsigned Integers	Optional	List of supported PIN Protocol versions.

All options are in the form key-value pairs with string IDs and boolean values. When an option is not present, the default is applied per table below. The following is a list of supported options:

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	resident key: Indicates that the device is capable of storing keys on the device itself and therefore can satisfy the authenticatorGetAssertion request with allowList parameter not specified or empty.	false
clientPin	Client PIN: 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. Client PIN is one of the ways to do user verification.	Not supported
up	user presence: Indicates that the device is capable of testing user presence.	true

uv	user verification: Indicates that the device is capable of verifying the user within itself. For example, devices with UI, biometrics fall into this category. If present and set to true, it indicates that the device is capable of user verification within itself and has been configured. If present and set to false, it indicates that the device is capable of user verification within itself and has not been yet configured. For example, a biometric device that has not yet been configured will return this parameter set to false. If absent, it indicates that the device is not capable of user verification within itself. A device that can only do Client PIN will not return the "uv" parameter. If a device is capable of verifying the user within itself as well as able to do Client PIN, it will return both "uv" and the Client PIN option.	Not Supported

5.5. authenticatorClientPIN (0x06)§

One of the design goals of this command is to have minimum burden on the authenticator and to not send actual encrypted PIN to the authenticator in normal authenticator usage scenarios to have more security. Hence, below design only sends PIN in encrypted format while setting or changing a PIN. On normal PIN usage scenarios, design uses randomized pinToken which gets generated every power cycle.

This command is used by the platform to establish key agreement with authenticator and getting sharedSecret, setting a new PIN on the authenticator, changing existing PIN on the authenticator and getting "pinToken" from the authenticator which can be used in subsequent authenticatorMakeCredential and authenticatorGetAssertion operations.

It takes the following input parameters:

Parameter name	Data type	Required?	Definition
pinProtocol (0x01)	Unsigned Integer	Required	PIN protocol version chosen by the client. For this version of the spec, this SHALL be the number 1.
subCommand (0x02)	Unsigned Integer	Required	The authenticator Client PIN sub command currently being requested
keyAgreement (0x03)	COSE_Key	Optional	Public key of platformKeyAgreementKey. The 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.
pinAuth (0x04)	Byte Array	Optional	First 16 bytes of HMAC-SHA-256 of encrypted contents using sharedSecret. See Setting a new PIN, Changing existing PIN and Getting pinToken from the authenticator for more details.
newPinEnc (0x05)	Byte Array	Optional	Encrypted new PIN using sharedSecret. Encryption is done over UTF-8 representation of new PIN.
pinHashEnc (0x06)	Byte Array	Optional	Encrypted first 16 bytes of SHA-256 of PIN using

sharedSecret.

The list of sub commands for PIN Protocol Version 1 is:

subCommand Name	subCommand Number
getRetries	0x01
getKeyAgreement	0x02
setPIN	0x03
changePIN	0x04
getPINToken	0x05

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

Parameter name	Data type	Required?	Definition
KeyAgreement (0x01)	COSE_Key	Optional	Authenticator key agreement public key in COSE_Key format. This will be used to establish a sharedSecret between platform and the authenticator. The 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.
pinToken (0x02)	Byte Array	Optional	Encrypted pinToken using sharedSecret to be used in subsequent authenticatorMakeCredential and authenticatorGetAssertion operations.
retries (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 Getting pinToken from the authenticator flows.

5.5.1. Client PIN Support Requirements§

- Platform has to fulfill following PIN support requirements while gathering input from the user:
  - Minimum PIN Length: 4 Unicode characters
  - Maximum PIN Length: UTF-8 representation must not exceed 63 bytes
- Authenticator has to fulfill following PIN support requirements:
  - Minimum PIN Length: 4 bytes
  - Maximum PIN Length: 63 bytes
  - Maximum consecutive incorrect PIN attempts: 8
    - retries counter represents the number of attempts left before PIN is blocked.
    - Each correct PIN entry resets the retries counter back to 8 unless the PIN is already blocked.
    - Each incorrect PIN entry decrements the retries by 1.
    - Once the retries counter reaches 0, the authenticator has to be reset before any further operations can happen that require a PIN.

- PIN storage on the device has to be of the same or better security assurances as of private keys on the device.

> Note: Authenticators can implement minimum PIN lengths that are longer than 4 bytes.

5.5.2. Authenticator Configuration Operations Upon Power Up§

Authenticator generates following configuration at power up. This is to have less burden on the authenticator as key agreement is an expensive operation. This also ensures randomness across power cycles.

Following are the operations authenticator performs on each powerup:

- Generate **"authenticatorKeyAgreementKey"**:
  - Generate an ECDH P-256 key pair called "authenticatorKeyAgreementKey" denoted by `(a, aG)` where "a" denotes the private key and "aG" denotes the public key.
    - See [RFC6090] Section 4.1 and [SP800-56A] for more ECDH key agreement protocol details.
- Generate **"pinToken"**:
  - Generate a random integer of length which is multiple of 16 bytes (AES block length).
  - "pinToken" is used so that there is minimum burden on the authenticator and platform does not have to not send actual encrypted PIN to the authenticator in normal authenticator usage scenarios. This also provides more security as we are not sending actual PIN even in encrypted form. "pinToken" will be given to the platform upon verification of the PIN to be used in subsequent authenticatorMakeCredential and authenticatorGetAssertion operations.

5.5.3. Getting Retries from Authenticator§

Retries count is the number of attempts remaining before lockout. When the device is nearing authenticator lockout, the platform can optionally warn the user to be careful while entering the PIN.

Platform performs the following operations to get retries:

- Platform sends authenticatorClientPIN command with following parameters to the authenticator:
  - pinProtocol: 0x01
  - subCommand: getRetries(0x01)
- Authenticator responds back with retries.

5.5.4. Getting sharedSecret from Authenticator§

Platform does the ECDH key agreement to arrive at sharedSecret to be used only during that transaction. 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:

- Platform sends authenticatorClientPIN command with following parameters to the authenticator:
  - pinProtocol: 0x01
  - subCommand: getKeyAgreement(0x02)
- Authenticator responds back with public key of authenticatorKeyAgreementKey, "aG".
- Platform generates **"platformKeyAgreementKey"**:

- Platform generates ECDH P-256 key pair called "platformKeyAgreementKey" denoted by `(b, bG)` where `"b"` denotes the private key and `"bG"` denotes the public key.

- Platform generates **"sharedSecret"**
  - Platform generates "sharedSecret" using SHA-256 over ECDH key agreement protocol using private key of platformKeyAgreementKey, "b" and public key of authenticatorKeyAgreementKey, "aG": `SHA-256((baG).x)`.
    - SHA-256 is done over only "x" curve point of baG.
    - See [RFC6090] Section 4.1 and appendix (C.2) of [SP800-56A] for more ECDH key agreement protocol details and key representation.

5.5.5. Setting a New PIN§

Following operations are performed to set up a new PIN:

- Platform gets sharedSecret from the authenticator.
- Platform collects new PIN ("newPinUnicode") from the user in Unicode format.
  - Platform checks the Unicode character length of "newPinUnicode" against the minimum 4 Unicode character requirement and returns CTAP2_ERR_PIN_POLICY_VIOLATION if the check fails.
  - Let "newPin" be the UTF-8 representation of "newPinUnicode".
    - Platform checks the byte length of "newPin" against the max UTF-8 representation limit of 63 bytes and returns CTAP2_ERR_PIN_POLICY_VIOLATION if the check fails.
- Platform sends §5.5 authenticatorClientPIN (0x06) command with following parameters to the authenticator:
  - pinProtocol: 0x01.
  - subCommand: setPIN(0x03).
  - keyAgreement: public key of platformKeyAgreementKey, "bG".
  - newPinEnc: Encrypted newPin using sharedSecret: `AES256-CBC(sharedSecret, IV=0, newPin)`.
    - During encryption, newPin is padded with trailing 0x00 bytes and is of minimum 64 bytes length. This is to prevent leak of PIN length while communicating to the authenticator. There is no PKCS #7 padding used in this scheme.
  - pinAuth: `LEFT(HMAC-SHA-256(sharedSecret, newPinEnc), 16)`.
    - The platform sends the first 16 bytes of the HMAC-SHA-256 result.
- Authenticator performs following operations upon receiving the request:
  - If Authenticator does not receive mandatory parameters for this command, it returns CTAP2_ERR_MISSING_PARAMETER error.
  - If a PIN has already been set, authenticator returns CTAP2_ERR_PIN_AUTH_INVALID error.
  - Authenticator generates "sharedSecret": `SHA-256((abG).x)` using private key of authenticatorKeyAgreementKey, "a" and public key of platformKeyAgreementKey, "bG".
    - SHA-256 is done over only `"x"` curve point of `"abG"`
    - See [RFC6090] Section 4.1 and appendix (C.2) of [SP800-56A] for more ECDH key agreement protocol details and key representation.
  - Authenticator verifies pinAuth by generating `LEFT(HMAC-SHA-256(sharedSecret, newPinEnc), 16)` and matching against input pinAuth parameter.
    - If pinAuth verification fails, authenticator returns CTAP2_ERR_PIN_AUTH_INVALID error.

- Authenticator decrypts newPinEnc using above "sharedSecret" producing newPin and checks newPin length against minimum PIN length of 4 bytes.
  - The decrypted padded newPin should be of at least 64 bytes length and authenticator determines actual PIN length by looking for first 0x00 byte which terminates the PIN.
  - If minimum PIN length check fails, authenticator returns CTAP2_ERR_PIN_POLICY_VIOLATION error.
  - Authenticator may have additional constraints for PIN policy. The current spec only enforces minimum length of 4 bytes.
- Authenticator stores `LEFT(SHA-256(newPin), 16)` on the device, sets the retries counter to 8, and returns CTAP2_OK.

5.5.6. Changing existing PIN§

Following operations are performed to change an existing PIN:

- Platform gets sharedSecret from the authenticator.
- Platform collects current PIN ("curPinUnicode") and new PIN ("newPinUnicode") from the user.
  - Platform checks the Unicode character length of "newPinUnicode" against the minimum 4 Unicode character requirement and returns CTAP2_ERR_PIN_POLICY_VIOLATION if the check fails.
  - Let "curPin" be the UTF-8 representation of "curPinUnicode" and "newPin" be the UTF-8 representation of "newPinUnicode"
    - Platform checks the byte length of "curPin" and "newPin" against the max UTF-8 representation limit of 63 bytes and returns CTAP2_ERR_PIN_POLICY_VIOLATION if the check fails.
- Platform sends authenticatorClientPIN command with following parameters to the authenticator:
  - pinProtocol: 0x01.
  - subCommand: changePIN(0x04).
  - keyAgreement: public key of platformKeyAgreementKey, "bG".
  - pinHashEnc: Encrypted first 16 bytes of SHA-256 hash of curPin using sharedSecret: `AES256-CBC(sharedSecret, IV=0, LEFT(SHA-256(curPin),16))`.
  - newPinEnc: Encrypted "newPin" using sharedSecret: `AES256-CBC(sharedSecret, IV=0, newPin)`.
    - During encryption, newPin is padded with trailing 0x00 bytes and is of minimum 64 bytes length. This is to prevent leak of PIN length while communicating to the authenticator. There is no PKCS #7 padding used in this scheme.
  - pinAuth: `LEFT(HMAC-SHA-256(sharedSecret, newPinEnc || pinHashEnc), 16)`.
    - The platform sends the first 16 bytes of the HMAC-SHA-256 result.
- Authenticator performs following operations upon receiving the request:
  - If Authenticator does not receive mandatory parameters for this command, it returns CTAP2_ERR_MISSING_PARAMETER error.
  - If the retries counter is 0, return CTAP2_ERR_PIN_BLOCKED error.
  - Authenticator generates "sharedSecret": SHA-256((abG).x) using private key of authenticatorKeyAgreementKey, "a" and public key of platformKeyAgreementKey, "bG".
    - SHA-256 is done over only `"x"` curve point of `"abG"`
    - See [RFC6090] Section 4.1 and appendix (C.2) of [SP800-56A] for more ECDH key agreement protocol details and key representation.

- Authenticator verifies pinAuth by generating `LEFT(HMAC-SHA-256(sharedSecret, newPinEnc || pinHashEnc), 16)` and matching against input pinAuth parameter.
  - If pinAuth verification fails, authenticator returns CTAP2_ERR_PIN_AUTH_INVALID error.
- Authenticator decrements the retries counter by 1.
- Authenticator decrypts pinHashEnc and verifies against its internal stored`LEFT(SHA-256(curPin), 16)`.
  - If a mismatch is detected, the authenticator performs the following operations:
    - Authenticator generates a new **"authenticatorKeyAgreementKey"**.
      - Generate a new ECDH P-256 key pair called "authenticatorKeyAgreementKey" denoted by `(a, aG)`, where `"a"` denotes the private key and `"aG"` denotes the public key.
        - See [RFC6090] Section 4.1 and [SP800-56A] for more ECDH key agreement protocol details.
    - Authenticator returns errors according to following conditions:
      - If the retries 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 retries counter to 8.
- Authenticator decrypts newPinEnc using above "sharedSecret" producing newPin and checks newPin length against minimum PIN length of 4 bytes.
  - The decrypted padded newPin should be of at least 64 bytes length and authenticator determines actual PIN length by looking for first 0x00 byte which terminates the PIN.
  - If minimum PIN length check fails, authenticator returns CTAP2_ERR_PIN_POLICY_VIOLATION error.
  - Authenticator may have additional constraints for PIN policy. The current spec only enforces minimum length of 4 bytes.
- Authenticator stores `LEFT(SHA-256(newPin), 16)` on the device and returns CTAP2_OK.

5.5.7. Getting pinToken from the Authenticator§

This step only has to be performed once for the lifetime of the authenticator/platform handle. Getting pinToken once provides allows high security without any additional roundtrips every time (except for the first key-agreement phase) and its overhead is minimal.

Following operations are performed to get pinToken which will be used in subsequent authenticatorMakeCredential and authenticatorGetAssertion operations:

- Platform gets sharedSecret from the authenticator.
- Platform collects PIN from the user.
- Platform sends authenticatorClientPIN command with following parameters to the authenticator:
  - pinProtocol: 0x01.
  - subCommand: getPinToken(0x05).
  - keyAgreement: public key of platformKeyAgreementKey, "bG".

- pinHashEnc: `AES256-CBC(sharedSecret, IV=0, LEFT(SHA-256(PIN),16))`.

- Authenticator performs following operations upon receiving the request:
  - If Authenticator does not receive mandatory parameters for this command, it returns CTAP2_ERR_MISSING_PARAMETER error.
  - If the retries counter is 0, return CTAP2_ERR_PIN_BLOCKED error.
  - Authenticator generates "sharedSecret": `SHA-256((abG).x)` using private key of authenticatorKeyAgreementKey, "a" and public key of platformKeyAgreementKey, "bG".
    - SHA-256 is done over only `"x"` curve point of `"abG"`
      - See [RFC6090] Section 4.1 and appendix (C.2) of [SP800-56A] for more ECDH key agreement protocol details and key representation.
  - Authenticator decrements the retries counter by 1.
  - Authenticator decrypts pinHashEnc and verifies against its internal stored `LEFT(SHA-256(curPin), 16)`.
    - If a mismatch is detected, the authenticator performs the following operations:
      - Authenticator generates a new **"authenticatorKeyAgreementKey"**.
        - Generate a new ECDH P-256 key pair called "authenticatorKeyAgreementKey" denoted by `(a, aG)`, where `"a"` denotes the private key and `"aG"` denotes the public key.
          - See [RFC6090] Section 4.1 and [SP800-56A] for more ECDH key agreement protocol details.
      - Authenticator returns errors according to following conditions:
        - If the retries 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 retries counter to 8.
  - Authenticator returns encrypted pinToken using "sharedSecret": `AES256-CBC(sharedSecret, IV=0, pinToken)`.
    - pinToken should be a multiple of 16 bytes (AES block length) without any padding or IV. There is no PKCS #7 padding used in this scheme.

5.5.8. Using pinToken§

Platform has the flexibility to manage the lifetime of pinToken based on the scenario however it should get rid of the pinToken as soon as possible when not required. Authenticator also can expire pinToken based on certain conditions like changing a PIN, timeout happening on authenticator, machine waking up from a suspend state etc. If pinToken has expired, authenticator will return CTAP2_ERR_PIN_TOKEN_EXPIRED and platform can act on the error accordingly.

5.5.8.1. Using pinToken in authenticatorMakeCredential§

Following operations are performed to use pinToken in authenticatorMakeCredential API:

- Platform gets pinToken from the authenticator.

- Platform sends authenticatorMakeCredential command with following additional optional parameter:
  - pinProtocol: 0x01.
  - pinAuth: `LEFT(HMAC-SHA-256(pinToken, clientDataHash), 16)`.
    - The platform sends the first 16 bytes of the HMAC-SHA-256 result.
- Authenticator verifies pinAuth by generating `LEFT(HMAC-SHA-256(pinToken, clientDataHash), 16)` and matching against input pinAuth parameter.
  - If pinAuth verification fails, authenticator returns CTAP2_ERR_PIN_AUTH_INVALID error.
  - If authenticator sees 3 consecutive mismatches, it returns CTAP2_ERR_PIN_AUTH_BLOCKED indicating that power recycle 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.
- Authenticator returns authenticatorMakeCredential response with "uv" bit set to 1.

If platform sends zero length pinAuth, authenticator needs to wait for user touch and then returns either CTAP2_ERR_PIN_NOT_SET if pin is not set or CTAP2_ERR_PIN_INVALID if pin has been set. This is done for the case where multiple authenticators are attached to the platform and the platform wants to enforce clientPin semantics, but the user has to select which authenticator to send the pinToken to.

5.5.8.2. Using pinToken in authenticatorGetAssertion

Following operations are performed to use pinToken in authenticatorGetAssertion API:

- Platform gets pinToken from the authenticator.
- Platform sends authenticatorGetAssertion command with following additional optional parameter:
  - pinProtocol: 0x01.
  - pinAuth: `LEFT(HMAC-SHA-256(pinToken, clientDataHash), 16)`.
- Authenticator verifies pinAuth by generating `LEFT(HMAC-SHA-256(pinToken, clientDataHash), 16)` and matching against input pinAuth parameter.
  - If pinAuth verification fails, authenticator returns CTAP2_ERR_PIN_AUTH_INVALID error.
  - If authenticator sees 3 consecutive mismatches, it returns CTAP2_ERR_PIN_AUTH_BLOCKED indicating that power recycle 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.
- Authenticator returns authenticatorGetAssertion response with "uv" bit set to 1.

If platform sends zero length pinAuth, authenticator needs to wait for user touch and then returns either CTAP2_ERR_PIN_NOT_SET if pin is not set or CTAP2_ERR_PIN_INVALID if pin has been set. This is done for the case where multiple authenticators are attached to the platform and the platform wants to enforce clientPin semantics, but the user has to select which authenticator to send the pinToken to.

5.5.8.3. Without pinToken in authenticatorGetAssertion

Following operations are performed without using pinToken in authenticatorGetAssertion API:

- Platform sends authenticatorGetAssertion command without pinAuth optional parameter.
- Authenticator returns authenticatorGetAssertion response with "uv" bit set to 0.

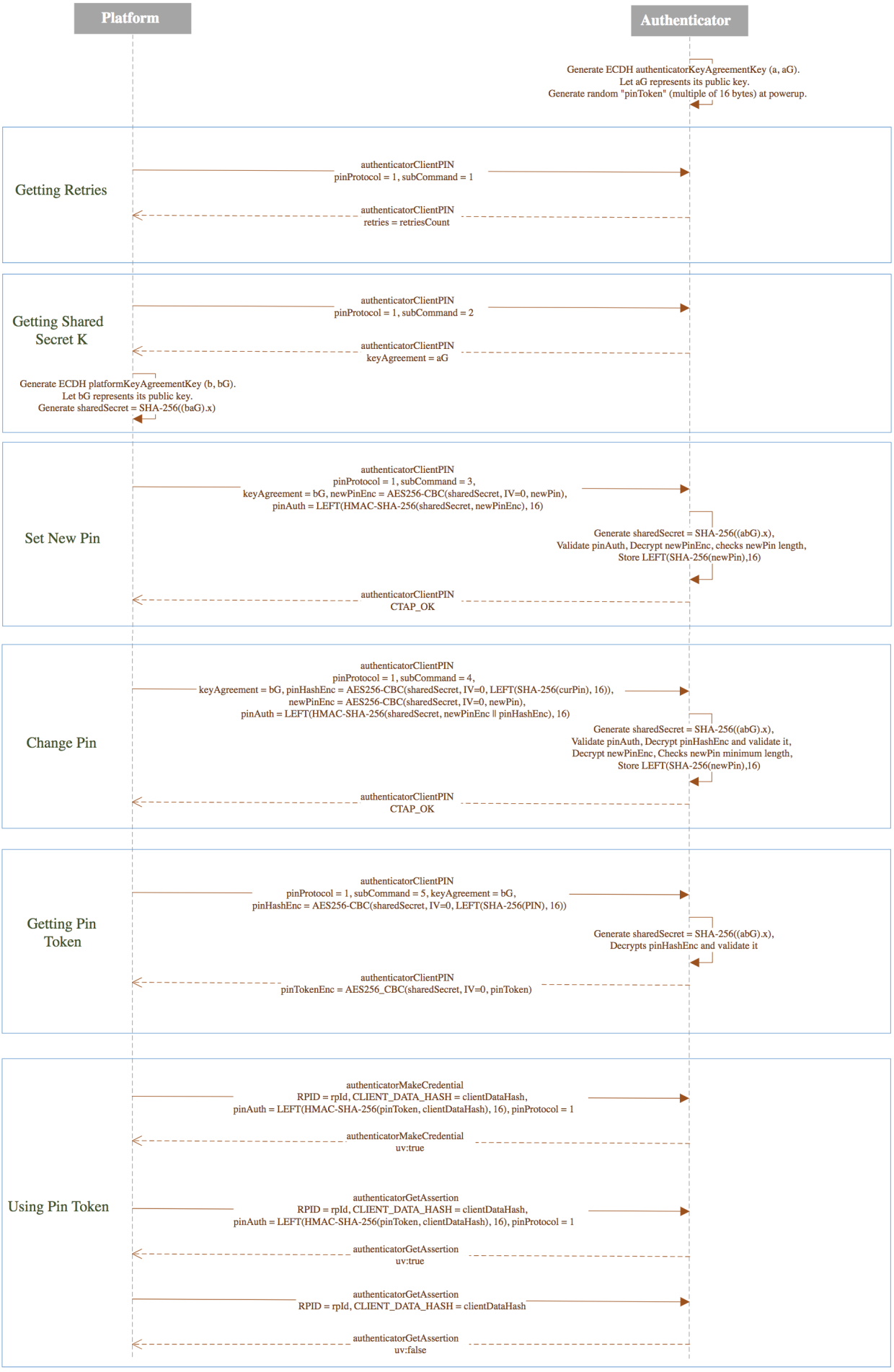

Figure 1 *Client PIN*

5.6. authenticatorReset (0x07)§

This method is used by the client to reset an authenticator back to a factory default state, invalidating all generated credentials. In order to prevent accidental trigger of this mechanism, some form of user approval MAY be performed on the authenticator itself, meaning that the client will have to poll the device until the reset has been performed. The actual user-flow to perform the reset will vary depending on the authenticator and it outside the scope of this specification.

6. 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 [RFC7049].

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 canonicalization suggested CTAP2 canonical CBOR encoding form as specified in Section 3.9 of[RFC7049]. 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.
- 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 2.4 in[RFC7049] 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 "replies". 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.

6.1. Commands§

All commands are structured as:

Name	Length	Required?	Definition
Command Value	1 byte	Required	The value of the command to execute
Command Parameters	variable	Optional	CBOR [RFC7049] encoded set of parameters. Some commands have parameters, while others do not (see below)

The assigned values for commands and their descriptions are:

Command Name	Command Value	Has parameters?
authenticatorMakeCredential	0x01	yes
authenticatorGetAssertion	0x02	yes
authenticatorGetInfo	0x04	no
authenticatorClientPIN	0x06	yes
authenticatorReset	0x07	no
authenticatorGetNextAssertion	0x08	no
authenticatorVendorFirst	0x40	NA
authenticatorVendorLast	0xBF	NA

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 [RFC7049]. 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.

EXAMPLE 1

A PublicKeyCredentialRpEntity DOM object defined as follows:

```
var rp = {
      name: "Acme"
    };
```

would be CBOR encoded as follows:

```
a1                                              # map(1)
    64                                          # text(4)
        6e616d65                                # "name"
    64                                          # text(4)
        41636d65                                # "Acme"
```

EXAMPLE 2

A PublicKeyCredentialUserEntity DOM object defined as follows:

```
var user = {
      id: Uint8Array.from(window.atob("MIIBkzCCATigAwIBAjCCAZMwggE4oAMCAQIwggGTMII="), c=>c
charCodeAt(0)),
      icon: "https://pics.example.com/00/p/aBjjjpqPb.png",
      name: "johnpsmith@example.com",
      displayName: "John P. Smith"
    };
```

would be CBOR encoded as follows:

```
a4                                              # map(4)
    62                                          # text(2)
        6964                                    # "id"
    58 20                                       # bytes(32)
        3082019330820138a003020102              # userid
        3082019330820138a003020102              # ...
        308201933082                            # ...
    64                                          # text(4)
        69636f6e                                # "icon"
    782b                                        # text(43)
        68747470733a2f2f706963732e657861        # "https://pics.example.com/00/p/aBjjjpqPb.
ng"
        6d706c652e636f6d2f30302f702f6142        # ...
        6a6a6a707150622e706e67                  # ...
    64                                          # text(4)
        6e616d65                                # "name"
    76                                          # text(22)
        6a6f686e70736d697468406578616d70        # "johnpsmith@example.com"
        6c652e636f6d                            # ...
    6b                                          # text(11)
        646973706c61794e616d65                  # "displayName"
    6d                                          # text(13)
        4a6f686e20502e20536d697468              # "John P. Smith"
```

EXAMPLE 3

A DOM object that is a sequence of PublicKeyCredentialParameters defined as follows:

```
var pubKeyCredParams = [
    {
      type: "public-key",
      alg: -7 // "ES256" as registered in the IANA COSE Algorithms registry
    },
    {
      type: "public-key",
      alg: -257 // "RS256" as registered by WebAuthn
    }
  ];
```

would be CBOR encoded as:

```
82                                      # array(2)
    a2                                  # map(2)
        63                              # text(3)
            616c67                      # "alg"
        26                              # -7 (ES256)
        64                              # text(4)
            74797065                    # "type"
        6a                              # text(10)
            7075626C69632D6B6579        # "public-key"
    a2                                  # map(2)
        63                              # text(3)
            616c67                      # "alg"
        390100                          # -257 (RS256)
        64                              # text(4)
            74797065                    # "type"
        6a                              # text(10)
            7075626C69632D6B6579        # "public-key"
```

For each command that contains parameters, the parameter map keys and value types are specified below:

Command	Parameter Name	Key	Value type
authenticatorMakeCredential	clientDataHash	0x01	byte string (CBOR major type 2).
	rp	0x02	CBOR definite length map (CBOR major type 5).
	user	0x03	CBOR definite length map (CBOR major type 5).
	pubKeyCredParams	0x04	CBOR definite length array (CBOR major type 4) of CBOR definite length maps (CBOR major type 5).
	excludeList	0x05	CBOR definite length array (CBOR major type 4) of CBOR definite length maps (CBOR major type 5).
	extensions	0x06	CBOR definite length map (CBOR major type 5).
	options	0x07	CBOR definite length map (CBOR major type 5).

	pinAuth	0x08	byte string (CBOR major type 2).
	pinProtocol	0x09	PIN protocol version chosen by the client. For this version of the spec, this SHALL be the number 1.
authenticatorGetAssertion	rpId	0x01	UTF-8 encoded text string (CBOR major type 3).
	clientDataHash	0x02	byte string (CBOR major type 2).
	allowList	0x03	CBOR definite length array (CBOR major type 4) of CBOR definite length maps (CBOR major type 5).
	extensions	0x04	CBOR definite length map (CBOR major type 5).
	options	0x05	CBOR definite length map (CBOR major type 5).
	pinAuth	0x06	byte string (CBOR major type 2).
	pinProtocol	0x07	PIN protocol version chosen by the client. For this version of the spec, this SHALL be the number 1.
authenticatorClientPIN	pinProtocol	0x01	Unsigned Integer. (CBOR major type 0)
	subCommand	0x02	Unsigned Integer. (CBOR major type 0)
	keyAgreement	0x03	COSE_Key
	pinAuth	0x04	byte string (CBOR major type 2).
	newPinEnc	0x05	byte string (CBOR major type 2). It is UTF-8 representation of encrypted input PIN value.
	pinHashEnc	0x06	byte string (CBOR major type 2).

EXAMPLE 4

The following is a complete encoding example of the `authenticatorMakeCredential` command (using same account and crypto parameters as above) and the corresponding `authenticatorMakeCredential_Response` response:

```
01                                  # authenticatorMakeCredential command
a5                                  # map(5)
   01                               # unsigned(1) - clientDataHash
   58 20                            # bytes(32)
      687134968222ec17202e42505f8ed2b1  # h'687134968222ec17202e42505f8ed2b16ae22f16bb05b8
25db9e602645f141'
      6ae22f16bb05b88c25db9e602645f141  #
   02                               # unsigned(2) - rp
   a2                               # map(2)
      62                            # text(2)
         6964                       # "id"
      6b                            # text(11)
         6578616d706c652e636f6d     # "example.com"
      64                            # text(4)
```

```
      64                                # text(4)
         6e616d65                       # "name"
      64                                # text(4)
         41636d65                       # "Acme"
   03                                   # unsigned(3) - user
   a4                                   # map(4)
      62                                # text(2)
         6964                           # "id"
      58 20                             # bytes(32)
         3082019330820138a003020102     # userid
         3082019330820138a003020102     # ...
         308201933082                   # ...
      64                                # text(4)
         69636f6e                       # "icon"
      78 2b                             # text(43)
         68747470733a2f2f706963732e6578 # "https://pics.example.com/00/p/aBjjjpqPb.png"
         616d706c652e636f6d2f30302f702f #
         61426a6a6a707150622e706e67     #
      64                                # text(4)
         6e616d65                       # "name"
      76                                # text(22)
         6a6f686e70736d697468406578616d # "johnpsmith@example.com"
         706c652e636f6d                 # ...
      6b                                # text(11)
         646973706c61794e616d65         # "displayName"
      6d                                # text(13)
         4a6f686e20502e20536d697468     # "John P. Smith"
   04                                   # unsigned(4) - pubKeyCredParams
   82                                   # array(2)
      a2                                # map(2)
         63                             # text(3)
            616c67                      # "alg"
         26                             # -7 (ES256)
         64                             # text(4)
            74797065                    # "type"
         6a                             # text(10)
            7075626C69632D6B6579        # "public-key"
      a2                                # map(2)
         63                             # text(3)
            616c67                      # "alg"
         390100                         # -257 (RS256)
         64                             # text(4)
            74797065                    # "type"
         6a                             # text(10)
            7075626C69632D6B6579        # "public-key"
   07                                   # unsigned(7) - options
   a1                                   # map(1)
      62                                # text(2)
         726b                           # "rk"
      f5                                # primitive(21)
```

`authenticatorMakeCredential_Response` response:

```
00                                      # status = success
a3                                      # map(3)
   01                                   # unsigned(1)
   66                                   # text(6)
      7061636b6564                      # "packed"
   02                                   # unsigned(2)
   58 9a                                # bytes(154)
      c289c5ca9b0460f9346ab4e42d842743  # authData
      404d31f4846825a6d065be597a87051d  # ...
```

```
   410000000bf8a011f38c0a4d15800617  # ...
   111f9edc7d00108959cead5b5c48164e  # ...
   8abcd6d9435c6fa363616c6765455332  # ...
   353661785820f7c4f4a6f1d79538dfa4  # ...
   c9ac50848df708bc1c99f5e60e51b42a  # ...
   521b35d3b69a61795820de7b7d6ca564  # ...
   e70ea321a4d5d96ea00ef0e2db89dd61  # ...
   d4894c15ac585bd23684              # ...
03                                   # unsigned(3)
a3                                   # map(3)
   63                                # text(3)
      616c67                         # "alg"
   26                                # -7 (ES256)
   63                                # text(3)
      736967                         # "sig"
   58 47                             # bytes(71)
      3045022013f73c5d9d530e8cc15cc9 # signature...
      bd96ad586d393664e462d5f0561235 # ...
      e6350f2b728902210090357ff910cc # ...
      b56ac5b596511948581c8fddb4a2b7 # ...
      9959948078b09f4bdc6229         # ...
   63                                # text(3)
      783563                         # "x5c"
   81                                # array(1)
      59 0197                        # bytes(407)
         3082019330820138a003020102  # certificate...
         020900859b726cb24b4c29300a  # ...
         06082a8648ce3d040302304731  # ...
         0b3009060355504061302555331  # ...
         143012060355040a0c0b597562  # ...
         69636f20546573743122302006  # ...
         0355040b0c1941757468656e74  # ...
         696361746f7220417474657374  # ...
         6174696f6e301e170d31363132  # ...
         303431313535305a170d3236  # ...
         313230323131353530305a3047  # ...
         310b3009060355040613025553  # ...
         31143012060355040a0c0b5975  # ...
         6269636f205465737431223020  # ...
         060355040b0c1941757468656e  # ...
         74696361746f72204174746573  # ...
         746174696f6e3059301306072a  # ...
         8648ce3d020106082a8648ce3d  # ...
         03010703420004ad11eb0e8852  # ...
         e53ad5dfed86b41e6134a18ec4  # ...
         e1af8f221a3c7d6e636c80ea13  # ...
         c3d504ff2e76211bb44525b196  # ...
         c44cb4849979cf6f896ecd2bb8  # ...
         60de1bf4376ba30d300b300906  # ...
         03551d1304023000300a06082a  # ...
         8648ce3d040302034900304602  # ...
         2100e9a39f1b03197525f7373e  # ...
         10ce77e78021731b94d0c03f3f  # ...
         da1fd22db3d030e7022100c4fa  # ...
         ec3445a820cf43129cdb00aabe  # ...
         fd9ae2d874f9c5d343cb2f113d  # ...
         a23723f3                    # ...
```

EXAMPLE 5

The following is a complete encoding example of the `authenticatorGetAssertion` command and the corresponding `authenticatorGetAssertion_Response` response:

```
02                                  # authenticatorGetAssertion command
a4                                  # map(4)
   01                               # unsigned(1)
   6b                               # text(11)
      6578616d706c652e636f6d        # "example.com"
   02                               # unsigned(2)
   58 20                            # bytes(32)
      687134968222ec17202e42505f8ed2b1  # clientDataHash
      6ae22f16bb05b88c25db9e602645f141  # ...
   03                               # unsigned(3)
   82                               # array(2)
      a2                            # map(2)
         62                         # text(2)
            6964                    # "id"
         58 40                      # bytes(64)
            f22006de4f905af68a43942f02  # credential ID
            4f2a5ece603d9c6d4b3df8be08  # ...
            ed01fc442646d034858ac75bed  # ...
            3fd580bf9808d94fcbee82b9b2  # ...
            ef6677af0adcc35852ea6b9e    # ...
         64                         # text(4)
            74797065                # "type"
         6a                         # text(10)
            7075626C69632D6B6579    # "public-key"
      a2                            # map(2)
         62                         # text(2)
            6964                    # "id"
         58 32                      # bytes(50)
            03030303030303030303030303  # credential ID
            03030303030303030303030303  # ...
            03030303030303030303030303  # ...
            030303030303030303030303    # ...
         64                         # text(4)
            74797065                # "type"
         6a                         # text(10)
            7075626C69632D6B6579    # "public-key"
   05                               # unsigned(5)
   a1                               # map(1)
      62                            # text(2)
         7576                       # "uv"
      f5                            # true
```

`authenticatorGetAssertion_Response` response:

```
00                                  # status = success
a5                                  # map(5)
   01                               # unsigned(1) - Credential
   a2                               # map(2)
      62                            # text(2)
         6964                       # "id"
      58 40                         # bytes(64)
         f22006de4f905af68a43942f02  # credential ID
         4f2a5ece603d9c6d4b3df8be08  # ...
         ed01fc442646d034858ac75bed  # ...
         3fd580bf9808d94fcbee82b9b2  # ...
         ef6677af0adcc35852ea6b9e    # ...
      64                            # text(4)
         74797065                   # "type"
      6a                            # text(10)
         7075626C69632D6B6579       # "public-key"
   02                               # unsigned(2)
   58 25                            # bytes(37)
      625ddadf743f5727e66bba8c2e387922  # authData
```

```
   d1af43c503d9114a8fba104d84d02bfa  # ...
   0100000011                        # ...
03                                   # unsigned(3)
58 47                                # bytes(71)
   304502204a5a9dd39298149d904769b5  # signature
   1a451433006f182a34fbdf66de5fc717  # ...
   d75fb350022100a46b8ea3c3b933821c  # ...
   6e7f5ef9daae94ab47f18db474c74790  # ...
   eaabb14411e7a0                    # ...
04                                   # unsigned(4) - publicKeyCredentialUserEntity
a4                                   # map(4)
  62                                 # text(2)
     6964                            # "id"
   58 20                             # bytes(32)
     3082019330820138a003020102      # userid
     3082019330820138a003020102      # ...
     308201933082                    # ...
   64                                # text(4)
     69636f6e                        # "icon"
   782b                              # text(43)
     68747470733a2f2f706963732e6578  # "https://pics.example.com/00/p/aBjjjpqPb.png"
     616d706c652e636f6d2f30302f702f  # ...
     61426a6a6a707150622e706e67      # ...
  64                                 # text(4)
     6e616d65                        # "name"
  76                                 # text(22)
     6a6f686e70736d697468406578616d  # "johnpsmith@example.com"
     706c652e636f6d                  # ...
  6b                                 # text(11)
     646973706c61794e616d65          # "displayName"
  6d                                 # text(13)
     4a6f686e20502e20536d697468      # "John P. Smith"
05                                   # unsigned(5) - numberofCredentials
01                                   # unsigned(1)
```

6.2. Responses§

All responses are structured as:

Name	Length	Required?	Definition
Status	1 byte	Required	The status of the response. 0x00 means success; all other values are errors. See the table in the next section for valid values.
Response Data	variable	Optional	CBOR encoded set of values.

Response data is encoded using a CBOR map (CBOR major type 5). The CBOR map must be encoded using the definite length variant.

For each response message, the map keys and value types are specified below:

Response Message	Member Name	Key	Value type
authenticatorMakeCredential_Response	fmt	0x01	text string (CBOR major type 3).
			byte string (CBOR major

	authData	0x02	type 2).
	attStmt	0x03	definite length map (CBOR major type 5).
authenticatorGetAssertion_Response	credential	0x01	definite length map (CBOR major type 5).
	authData	0x02	byte string (CBOR major type 2).
	signature	0x03	byte string (CBOR major type 2).
	publicKeyCredentialUserEntity	0x04	definite length map (CBOR major type 5).
	numberOfCredentials	0x05	unsigned integer(CBOR major type 0).
authenticatorGetNextAssertion_Response	credential	0x01	definite length map (CBOR major type 5).
	authData	0x02	byte string (CBOR major type 2).
	signature	0x03	byte string (CBOR major type 2).
	publicKeyCredentialUserEntity	0x04	definite length map (CBOR major type 5).
authenticatorGetInfo_Response	versions	0x01	definite length array (CBOR major type 4) of UTF-8 encoded strings (CBOR major type 3).
	extensions	0x02	definite length array (CBOR major type 4) of UTF-8 encoded strings (CBOR major type 3).
	aaguid	0x03	byte string (CBOR major type 2). 16 bytes in length and encoded the same as MakeCredential AuthenticatorData, as specified in [WebAuthn].
	options	0x04	Definite length map (CBOR major type 5) of key-value pairs where keys are UTF8 strings (CBOR major type 3) and values are booleans (CBOR simple value 21).

	maxMsgSize	0x05	unsigned integer(CBOR major type 0). This is the maximum message size supported by the authenticator.
	pinProtocols	0x06	array of unsigned integers (CBOR major type). This is the list of pinProtocols supported by the authenticator.
authenticatorClientPIN_Response	keyAgreement	0x01	Authenticator public key in COSE_Key format. The 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.
	pinToken	0x02	byte string (CBOR major type 2).
	retries	0x03	Unsigned integer (CBOR major type 0). This is number of retries left before lockout.

6.3. 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.
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.
0x16	CTAP2_ERR_UNSUPPORTED_EXTENSION	Unsupported extension.
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.
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.
0x2A	CTAP2_ERR_NO_OPERATION_PENDING	No outstanding operations.
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	Timeout waiting for user interaction.
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,pinAuth, verification failed.

0x34	CTAP2_ERR_PIN_AUTH_BLOCKED	PIN authentication,pinAuth, blocked. Requires power recycle to reset.
0x35	CTAP2_ERR_PIN_NOT_SET	No PIN has been set.
0x36	CTAP2_ERR_PIN_REQUIRED	PIN is required for the selected operation.
0x37	CTAP2_ERR_PIN_POLICY_VIOLATION	PIN policy violation. Currently only enforces minimum length.
0x38	CTAP2_ERR_PIN_TOKEN_EXPIRED	pinToken expired on authenticator.
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.
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.

7. Interoperating with CTAP1/U2F authenticators§

This section defines how a platform maps CTAP2 requests to CTAP1/U2F requests and CTAP1/U2F responses to CTAP2 responses in order to support CTAP1/U2F authenticators via CTAP2. CTAP2 requests can be mapped to CTAP1/U2F requests provided the CTAP2 request does not have parameters that only CTAP2 authenticators can fulfill. The processes for RPs to use to verify CTAP1/U2F based authenticatorMakeCredential and authenticatorGetAssertion responses are also defined below. Platform may choose to skip this feature and work only with CTAP devices.

7.1. 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.

7.1.1. 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

7.1.2. 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.

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.

7.2. 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 array 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. Map the U2F registration response message (see: FIDO U2F Raw Message Formats v1.0 §registration-response-message-success) to a CTAP2 authenticatorMakeCredential response message:
  - Generate `authenticatorData` from the U2F registration response message (FIDO U2F Raw Message Formats v1.0 §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 array 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 ([RFC8152] Section 7).
      - Let `attestedCredData` be a byte array 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 array 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].
- Let `attestationObject` be a CBOR map (see "attObj" in 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`.

5. Return `attestationObject` to the caller.

EXAMPLE 6

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"},
 4: [{"type": "public-key", "alg": -7},
     {"type": "public-key", "alg": -257}]}
```

CTAP1/U2F Request from above CTAP2 authenticatorMakeCredential request

```
687134968222EC17202E42505F8ED2B16AE22F16BB05B88C25DB9E602645F141    # clientDataHash
1194228DA8FDBDEEFD261BD7B6595CFD70A50D70C6407BCF013DE96D4EFB17DE    # 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)
0500302E312C302A06035504031323597562696362F2055324620526F6F742043    # ...
412053657269616C20343537323030363313020170D3134303830313030303030    # ...
30305A180F323035303039303430303030305A302C312A302806035504030C    # ...
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

```
1194228DA8FDBDEEFD261BD7B6595CFD70A50D70C6407BCF013DE96D4EFB17DE    # 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'1194228DA8FDBDEEFD261BD7B6595CFD70A50D70C6407BCF013DE96D4EFB17DE
      410000000000000000000000000000000000000000000403EBD89BF77EC509755
      EE9C2635EFAAAC7B2B9C5CEF1736C3717DA48534C8C6B654D7FF945F50B5CC4E
      78055BDD396B64F78DA2C5F96200CCD415CD08FE420038A50102032620012158
      20E87625896EE4E46DC032766E8087962F36DF9DFE8B567F3763015B1990A60E
      1422582027DE612D66418BDA1950581EBC5C8C1DAD710CB14C22F8C97045F461
      2FB20C91',
 3: {"sig": h'30450220324779C68F3380288A1197B6095F7A6EB9B1B1C127F66AE12A99FE85
              32EC23B9022100E39516AC4D61EE64044D50B415A6A4D4D84BA6D895CB5AB7A1
              AA7D081DE341FA',
             "x5c": [h'3082024A30820132A0030201020204046C8822300D06092A864886F70D01010B
                      0500302E312C302A06035504031323597562696362F2055324620526F6F742043
                      412053657269616C20343537323030363313020170D3134303830313030303030
                      30305A180F323035303039303430303030305A302C312A302806035504030C
                      2159756269636F2055324620454520536572696 16C2032343931383233323437
                      373030593013060 72A8648CE3D020106082A8648CE3D030107034200043CCAB9
                      2CCB97287EE8E639437E21FCD6B6F165B2D5A3F3DB131D31C16B742BB476D8D1
```

```
E99080EB546C9BBDF556E6210FD42785899E78CC589EBE310F6CDB9FF4A33B30
39302206092B0601040182C40A020415312E332E362E312E342E312E34313438
322E312E323013060B2B0601040182E51C020101040403020430300D06092A86
4886F70D01010B050003820101009F9B052248BC4CF42CC5991FCAABAC9B651B
BE5BDCDC8EF0AD2C1C1FFB36D18715D42E78B249224F92C7E6E7A05C49F0E7E4
C881BF2E94F45E4A21833D7456851D0F6C145A29540C874F3092C934B43D222B
8962C0F410CEF1DB75892AF116B44A96F5D35ADEA3822FC7146F6004385BCB69
B65C99E7EB6919786703C0D8CD41E8F75CCA44AA8AB725AD8E799FF3A8696A6F
1B2656E631B1E40183C08FDA53FA4A8F85A05693944AE179A1339D002D15CABD
810090EC722EF5DEF9965A371D415D624B68A2707CAD97BCDD1785AF97E258F3
3DF56A031AA0356D8E8D5EBCADC74E071636C6B110ACE5CC9B90DFEACAE640FF
1BB0F1FE5DB4EFF7A95F060733F5']}}
```

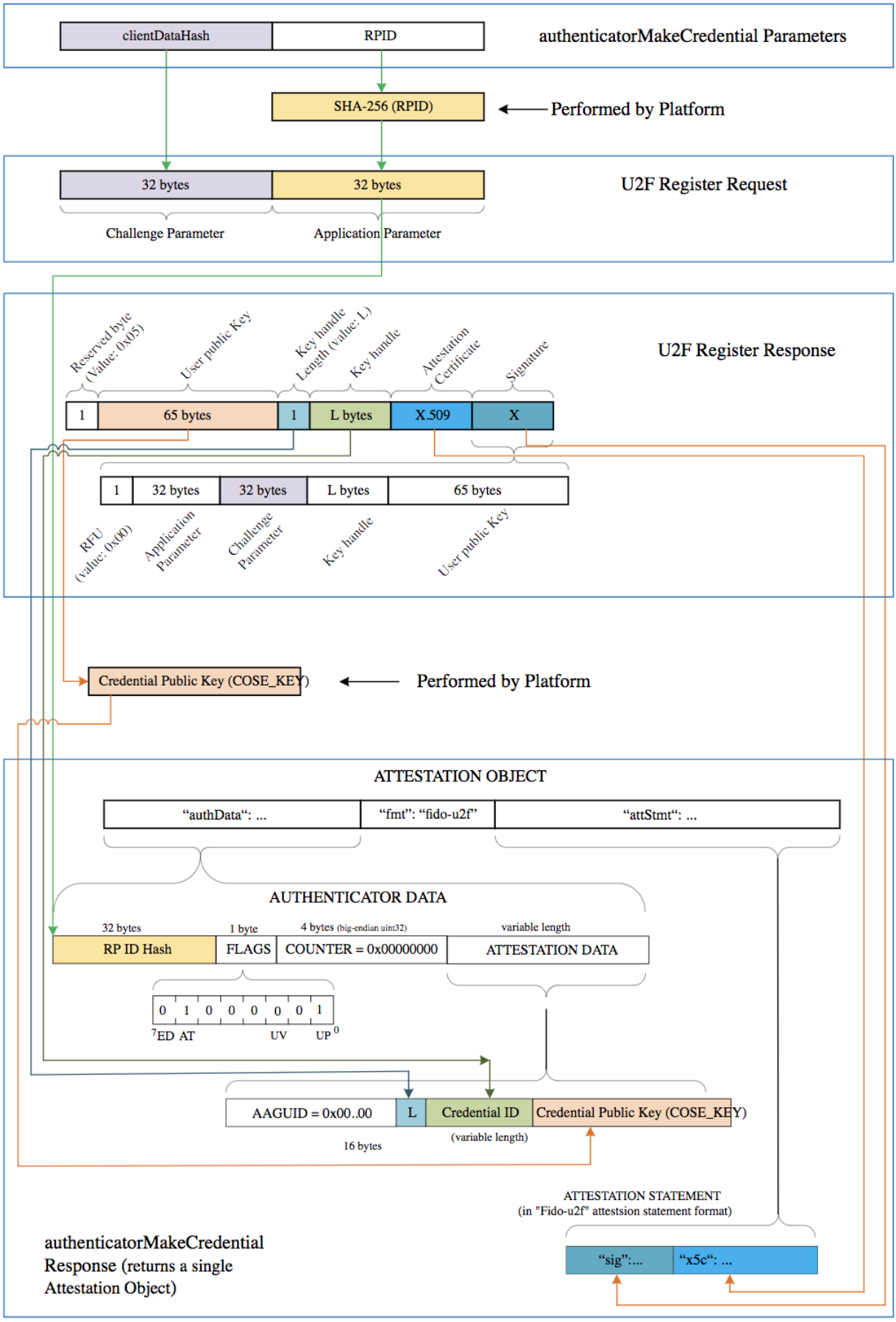

Figure 2 *Mapping: WebAuthn authenticatorMakeCredential to and from CTAP1/U2F Registration Messages.*

7.3. 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 array of size 32 initialized with SHA-256 hash of `rp.id` parameter as CTAP1/U2F application parameter (32 bytes).
        - Let `credentialId` is the byte array initialized with the id for this PublicKeyCredentialDescriptor.
        - Let `keyHandleLength` be a byte initialized with length of `credentialId` byte array.
        - Let `u2fAuthenticateRequest` be a byte array 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. 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 array 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.

EXAMPLE 7

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
1194228DA8FDBDEEFD261BD7B6595CFD70A50D70C6407BCF013DE96D4EFB17DE    # 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

```
1194228DA8FDBDEEFD261BD7B6595CFD70A50D70C6407BCF013DE96D4EFB17DE    # 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'1194228DA8FDBDEEFD261BD7B6595CFD70A50D70C6407BCF013DE96D4EFB17DE
      010000003B',
 3: h'304402207BDE0A52AC1F4C8B27E003A370CD66A4C7118DD22D5447835F45B99C
      68423FF702203C517B47877F85782DE10086A783D1E7DF4E3639E771F5F6AFA3
      5AAD5373858E'}
```

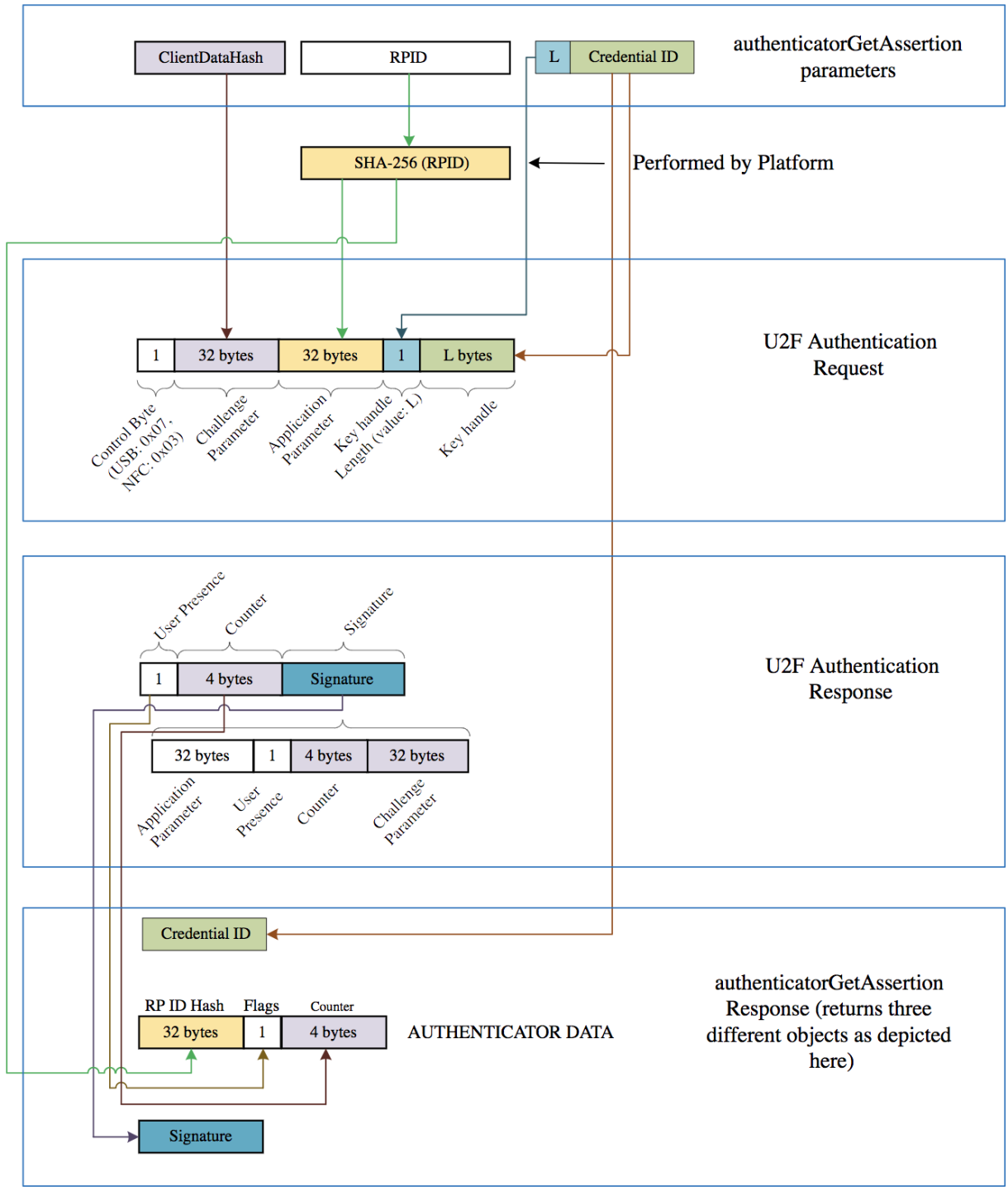

Figure 3 *Mapping: WebAuthn authenticatorGetAssertion to and from CTAP1/U2F Authentication Messages.*

8. Transport-specific Bindings§

8.1. USB Human Interface Device (USB HID)§

8.1.1. 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.

8.1.2. 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.

8.1.3. 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.

8.1.4. 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.

8.1.5. 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.

8.1.5.1. 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, no state is maintained in the device and a host application must assume that any other process may execute other transactions at any time.

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.

8.1.5.2. 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.

8.1.5.3. 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.

8.1.5.4. 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.

8.1.6. 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.

8.1.7. 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.

8.1.8. 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.

8.1.8.1. 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
bInterfaceSubClass	0x00	No interface subclass
bInterfaceProtocol	0x00	No interface protocol

Endpoint 1 descriptor

Mnemonic	Value	Description
bmAttributes	0x03	Interrupt transfer
bEndpointAdresss	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
bEndpointAdresss	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.

8.1.8.2. 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 8
// 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.

8.1.9. CTAPHID commands

The CTAPHID protocol implements the following commands.

8.1.9.1. 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.

8.1.9.1.1. 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

8.1.9.1.2. 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

8.1.9.1.3. 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

8.1.9.1.4. 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

8.1.9.1.5. 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

8.1.9.1.6. 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
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.

8.1.9.1.7. 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.

8.1.9.2. Optional commands§

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

8.1.9.2.1. 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

8.1.9.2.2. 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

8.1.9.3. 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).

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

8.2.1. Conformance§

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

8.2.2. 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.

8.2.3. Applet selection§

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.

8.2.4. 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.

8.2.4.1. 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

8.2.4.2. 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]

8.2.5. 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 9

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

```
01A80158206871349682222EC17202E42505F8ED2B16AE22F16BB05B88C25DB9E
602645F14102A262696469746573742E63746170646E616D6569746573742E63
74617003A362696458202B6689BB18F4169F069FBCDF50CB6EA3C60A861B9A7B
63946983E0B577B78C70646E616D65717465737463746170406374617 02E636F
6D6B646973706C61794E616D6569546573742043746170048 3A263616C672664
747970656A7075626C69632D6B6579A263616C6739010064747970656A707562
6C69632D6B6579A263616C6738246474797065 6A7075626C69632D6B657906A1
6B686D61632D736563726574F507A162726BF50850FC43AAA411D948CC6C3706
8B8DA1D5080901
```

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

- APDU command 1:

```
Platform Request:
90 10 00 00
F0
01A80158206871349682222EC17202E42505F8ED2B16AE22F16BB05B88C25DB9E
602645F14102A262696469746573742E63746170646E616D6569746573742E63
74617003A362696458202B6689BB18F4169F069FBCDF50CB6EA3C60A861B9A7B
63946983E0B577B78C70646E616D65717465737463746170406374617 02E636F
6D6B646973706C61794E616D6569546573742043746170048 3A263616C672664
747970656A7075626C69632D6B6579A263616C6739010064747970656A707562
6C69632D6B6579A263616C6738246474797065 6A7075626C69632D6B657906A1
6B686D61632D736563726574F507A162

Authenticator Response:
9000
```

- APDU command 2:

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

Authenticator Response:
```

```
00
A301667061636B6564025900A20021F5FC0B85CD22E60623BCD7D1CA48948909
249B4776EB515154E57B66AE12C500000055F8A011F38C0A4D15800617111F9E
DC7D0010F4D57B23DD0CB785680CDAA7F7E44F60A5010203262001215820DF01
7D0B286795BEA153D166A0A15B4F6B67A3AF4A101E10E8496F3DD3C5D1A92258
2094B22551E6325D7733C41BB2F5A642ADEE417C97E0906197B5B0CD8B8D6C6B
A7A16B686D61632D736563726574F503A363616C67266373696758473045022 0
7CCAC57A1E43DF24B0847EEBF119D28DCDC5048F7DCD8EDD79E79721C41BCF2D
022100D89EC75B92CE8FF9E46FE7F8C87995694A63E5B78AB85C47B9DA
6100
```

- APDU command 3:

```
Platform Request:
80 C0 00 00 00

Authenticator Response:
1C580A8EC83A6378356381590197308201933082013 8A003020102020900859B
726CB24B4C29300A06082A8648CE3D0403023047310B30090603550406130255
5331143012060355040A0C0B59756269636F2054657374312230200606355040B
0C1941757468656E74696361746F722041747465737461746 96F6E301E170D31
363132303431313535303 05A170D323631323032313135353030 5A3047310B30
090603550406130255533114301206035504 0A0C0B59756269636F2054657374
312230200606355040B0C1941757468656E74696361746F722041747465737461
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.

8.2.6. Commands§

8.2.6.1. 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.

8.2.6.2. 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.

8.3. Bluetooth Smart / Bluetooth Low Energy Technology§

8.3.1. 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].

8.3.2. 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.

8.3.3. 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.

8.3.4. 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.

8.3.4.1. 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 §6 Message Encoding.

8.3.4.2. 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
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 §6 Message Encoding.

8.3.4.3. 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.
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.

8.3.5. 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:

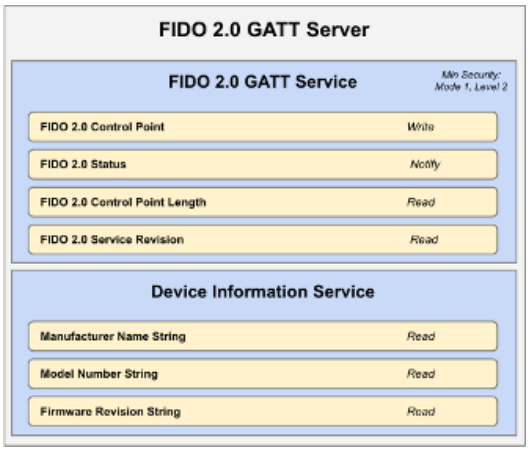

Figure 4 *Mandatory GATT services and characteristics that MUST be offered by a FIDO Authenticator. Note that the Generic Access Profile Service ([BTGAS]) is not present as it is already mandatory for any Bluetooth Low Energy Technology compliant device.*

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.

8.3.5.1. 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
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 a deprecated field that 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.

8.3.5.2. Device Information Service§

An authenticator SHALL implement the Device Information Service[BTDIS] with 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.

8.3.5.3. Generic Access Profile Service§

Every authenticator SHALL implement the Generic Access Profile Service[BTGAS] with the following characteristics:

- Device Name
- Appearance

8.3.6. 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.

8.3.7. 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.

8.3.8. 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.

8.3.9. 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.

8.3.10. 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 one 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]`

8.3.11. 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.

8.3.12. Implementation Considerations§

8.3.12.1. Bluetooth pairing: Client considerations

As noted in §8.3.2 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.

8.3.12.2. 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].

8.3.13. 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.
- 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

8.3.14. 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.

8.3.15. 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.

8.3.16. 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.

9. Defined Extensions§

This section defines an authenticator extension and corresponding WebAuthn extension.

9.1. 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.

Client extension input

`create()` : A boolean value to indicate that this extension is requested by the Relying Party.

```
partial dictionary AuthenticationExtensionsClientInputs {
  bool 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 {
  bool 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},
  >   7: {"rk": true}
  > }
  > ```

  - The authenticator generates a random 32-byte value (called CredRandom) and associates it with the credential.
  - The authenticator responds with the following CBOR map entry in the "extensions" fields to the authenticator:
    - "hmac-secret": true

    > EXAMPLE 12
    > Sample "extensions" field value in the authenticatorData:
    >
    > ```
    > {"hmac-secret": true}
    > ```

- **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 platformKeyAgreementKey, "bG".
      - saltEnc(0x02): Encrypt one or two salts (Called salt1 (32 bytes) and salt2 (32 bytes)) using sharedSecret as follows:

- One salt case: `AES256-CBC(sharedSecret, IV=0, salt1 (32 bytes))`.
- Two salt case: `AES256-CBC(sharedSecret, IV=0, salt1 (32 bytes) || salt2 (32 bytes))`.

- saltAuth(0x03): `LEFT(HMAC-SHA-256(sharedSecret, saltEnc), 16)`.
  - The platform sends the first 16 bytes of the HMAC-SHA-256 result.

EXAMPLE 13

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:
  - The authenticator waits for user consent.
  - The authenticator generates "sharedSecret": `SHA-256((abG).x)` using the private key of authenticatorKeyAgreementKey, "a" and the public key of platformKeyAgreementKey, "bG".
    - SHA-256 is done over only the `"x"` curve point of `"abG"`.
    - See [RFC6090] Section 4.1 and Appendix (C.2) of [SP800-56A] for more ECDH key agreement protocol details and key representation information.
  - The authenticator verifies saltEnc by generating `LEFT(HMAC-SHA-256(sharedSecret, saltEnc), 16)` and matching against the input saltAuth parameter.
  - 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 sharedSecret as part of "extensions" parameter:
    - One salt case: `"hmac-secret": AES256-CBC(sharedSecret, IV=0, output1 (32 bytes))`
    - Two salt case: `"hmac-secret": AES256-CBC(sharedSecret, IV=0, output1 (32 bytes) || output2 (32 bytes))`

EXAMPLE 14

Sample "extensions" field value in the authenticatorData:

```
{ "hmac-secret": h'1F91526CAE456E4CBB71C4DDE7BB877157E6E54DFED3015D7D4DBB2269AFCDE6A91E
D267EBBF848EB95A68E79C7AC705E351D543DB0165887D6290FD47A40C4' }
```

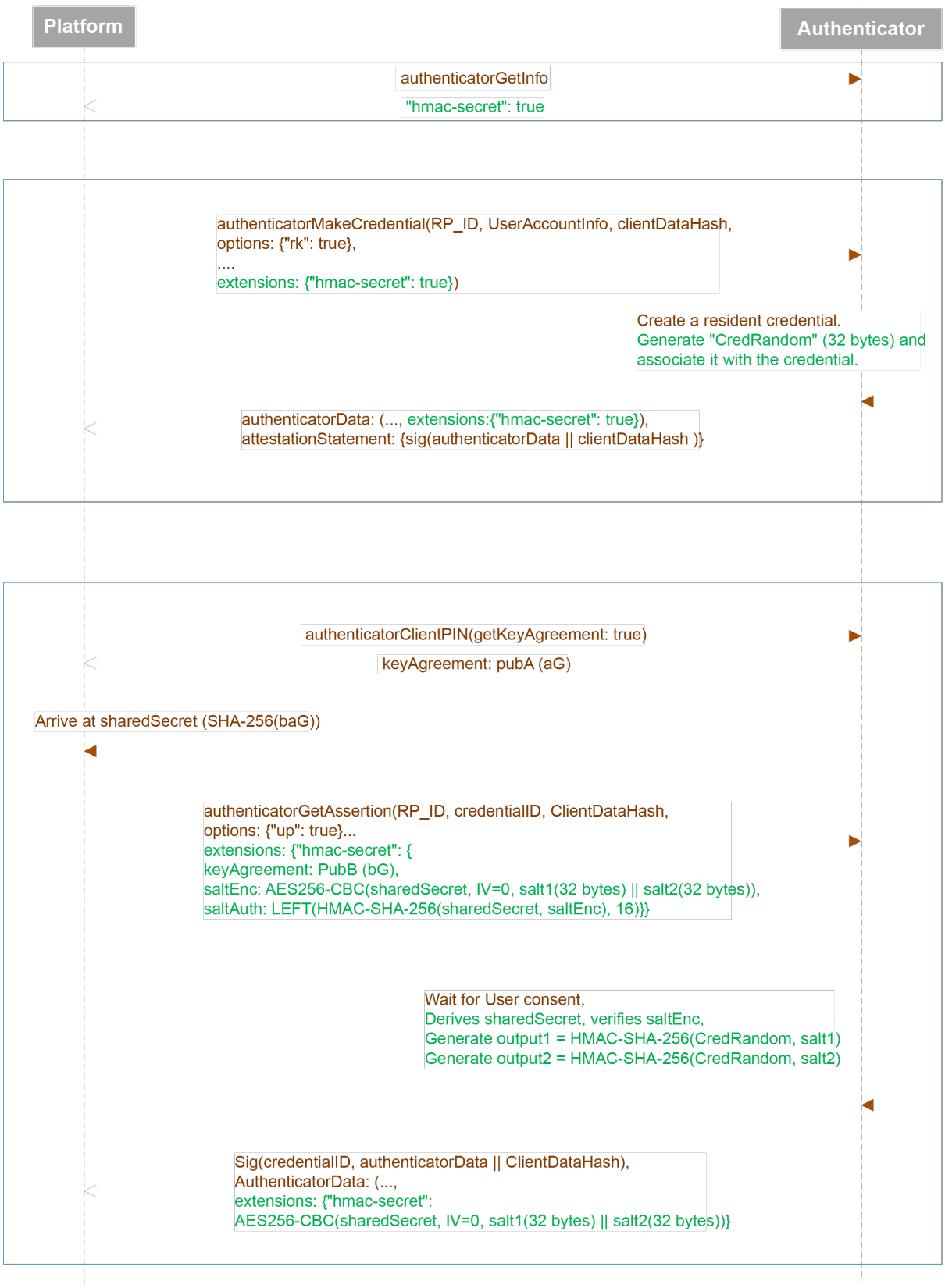

Figure 5 *hmac-secret*

Authenticator extension output

Same as the client extension output, except represented in CBOR.

10. IANA Considerations§

10.1. WebAuthn Extension Identifier Registrations§

This section registers the extension identifier values defined in Section §9 Defined Extensions in the IANA "WebAuthn Extension Identifier" registry.

- 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 §9.1 HMAC Secret Extension (hmac-secret) of this specification

11. Security Considerations§

See FIDO Security Reference document [FIDOSecRef].

Index§

Terms defined by this specification§

Terms defined by reference§

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

[CREDENTIAL-MANAGEMENT-1]
Mike West. Credential Management Level 1. 4 August 2017. WD. URL: https://www.w3.org/TR/credential-management-1/

[FIDOSecRef]
R. Lindemann; D. Baghdasaryan; B. Hill. FIDO Security Reference. Implementation Draft. URL: https://fidoalliance.org/specs/fido-v2.0-id-20180227/fido-security-ref-v2.0-id-20180227.html

[FIDOServerGuidelines]
FIDO2 Server Guidelines. URL: https://fidoalliance.org/specs/fido-v2.0-rd-20180702/fido-server-v2.0-rd-20180702.html

[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

[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

[RFC2397]
L. Masinter. The "data" URL scheme. August 1998. Proposed Standard. URL: https://tools.ietf.org/html/rfc2397

[RFC6090]
D. McGrew; K. Igoe; M. Salter. Fundamental Elliptic Curve Cryptography Algorithms. February 2011. Informational. URL: https://tools.ietf.org/html/rfc6090

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

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

[SEC1V2]
SEC1: Elliptic Curve Cryptography, Version 2.0. May 2009. URL: http://secg.org/download/aid-780/sec1-v2.pdf

[SP800-56A]
Elaine Barker; et al. Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography. May 2013. URL: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar2.pdf

[U2FBle]
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. FIDO U2F Raw Message Formats v1.0. 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; et al. Web Authentication: An API for accessing Public Key Credentials Level 1. March 2018. CR. URL: https://www.w3.org/TR/webauthn/

[WebIDL]
Cameron McCormack; Boris Zbarsky; Tobie Langel. Web IDL. 15 December 2016. ED. URL: https://heycam.github.io/webidl/

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

IDL Index§

```
partial dictionary AuthenticationExtensionsClientInputs {
  bool 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 {
  bool hmacCreateSecret;
};

dictionary HMACGetSecretOutput {
  required ArrayBuffer output1;
  ArrayBuffer output2;
};

partial dictionary AuthenticationExtensionsClientOutputs {
  HMACGetSecretOutput hmacGetSecret;
};
```

↑

→