

I n t e r n a t i o n a l T e l e c o m m u n i c a t i o n U n i o n

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

X.1278

(11/2018)

SERIES X: DATA NETWORKS, OPEN SYSTEM
COMMUNICATIONS AND SECURITY

Cyberspace security – Identity management

**Client to authenticator protocol/Universal
2-factor framework**

Recommendation ITU-T X.1278

ITU-T X-SERIES RECOMMENDATIONS

DATA NETWORKS, OPEN SYSTEM COMMUNICATIONS AND SECURITY

PUBLIC DATA NETWORKS	X.1–X.199
OPEN SYSTEMS INTERCONNECTION	X.200–X.299
INTERWORKING BETWEEN NETWORKS	X.300–X.399
MESSAGE HANDLING SYSTEMS	X.400–X.499
DIRECTORY	X.500–X.599
OSI NETWORKING AND SYSTEM ASPECTS	X.600–X.699
OSI MANAGEMENT	X.700–X.799
SECURITY	X.800–X.849
OSI APPLICATIONS	X.850–X.899
OPEN DISTRIBUTED PROCESSING	X.900–X.999
INFORMATION AND NETWORK SECURITY	
General security aspects	X.1000–X.1029
Network security	X.1030–X.1049
Security management	X.1050–X.1069
Telebiometrics	X.1080–X.1099
SECURE APPLICATIONS AND SERVICES (1)	
Multicast security	X.1100–X.1109
Home network security	X.1110–X.1119
Mobile security	X.1120–X.1139
Web security	X.1140–X.1149
Security protocols (1)	X.1150–X.1159
Peer-to-peer security	X.1160–X.1169
Networked ID security	X.1170–X.1179
IPTV security	X.1180–X.1199
CYBERSPACE SECURITY	
Cybersecurity	X.1200–X.1229
Countering spam	X.1230–X.1249
Identity management	X.1250–X.1279
SECURE APPLICATIONS AND SERVICES (2)	
Emergency communications	X.1300–X.1309
Ubiquitous sensor network security	X.1310–X.1319
Smart grid security	X.1330–X.1339
Certified mail	X.1340–X.1349
Internet of things (IoT) security	X.1360–X.1369
Intelligent transportation system (ITS) security	X.1370–X.1389
Distributed ledger technology security	X.1400–X.1429
Distributed ledger technology security	X.1430–X.1449
Security protocols (2)	X.1450–X.1459
CYBERSECURITY INFORMATION EXCHANGE	
Overview of cybersecurity	X.1500–X.1519
Vulnerability/state exchange	X.1520–X.1539
Event/incident/heuristics exchange	X.1540–X.1549
Exchange of policies	X.1550–X.1559
Heuristics and information request	X.1560–X.1569
Identification and discovery	X.1570–X.1579
Assured exchange	X.1580–X.1589
CLOUD COMPUTING SECURITY	
Overview of cloud computing security	X.1600–X.1601
Cloud computing security design	X.1602–X.1639
Cloud computing security best practices and guidelines	X.1640–X.1659
Cloud computing security implementation	X.1660–X.1679
Other cloud computing security	X.1680–X.1699

For further details, please refer to the list of ITU-T Recommendations.

Recommendation ITU-T X.1278

Client to authenticator protocol/Universal 2-factor framework

Summary

Recommendation ITU-T X.1278 describes an application layer protocol for communication between an external authenticator and another client/platform, as well as bindings of this application protocol to a variety of transport protocols using different physical media.

History

Edition	Recommendation	Approval	Study Group	Unique ID*
1.0	ITU-T X.1278	2018-11-29	17	11.1002/1000/13728

Keywords

Authentication, CTAP, identity, two-factor authentication, U2F.

* To access the Recommendation, type the URL <http://handle.itu.int/> in the address field of your web browser, followed by the Recommendation's unique ID. For example, <http://handle.itu.int/11.1002/1000/11830-en>.

FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications, information and communication technologies (ICTs). The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Assembly (WTSA), which meets every four years, establishes the topics for study by the ITU-T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSA Resolution 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

Compliance with this Recommendation is voluntary. However, the Recommendation may contain certain mandatory provisions (to ensure, e.g., interoperability or applicability) and compliance with the Recommendation is achieved when all of these mandatory provisions are met. The words "shall" or some other obligatory language such as "must" and the negative equivalents are used to express requirements. The use of such words does not suggest that compliance with the Recommendation is required of any party.

INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, ITU had not received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementers are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database at <http://www.itu.int/ITU-T/ipr/>.

© ITU 2019

All rights reserved. No part of this publication may be reproduced, by any means whatsoever, without the prior written permission of ITU.

Table of Contents

		Page
1	Scope.....	1
2	References.....	1
3	Definitions	1
4	Abbreviations and acronyms	1
5	Conventions	2
	5.1 Conformance	2
6	Overview.....	2
7	Relationship to other specifications	3
8	Protocol structure.....	3
9	Protocol overview	3
10	Authenticator API.....	4
	10.1 authenticatorMakeCredential (0x01).....	4
	10.2 authenticatorGetAssertion(0x02)	7
	10.3 authenticatorGetNextAssertion (0x08).....	10
	10.4 Client logic	11
	10.5 authenticatorCancel (0x03)	11
	10.6 authenticatorGetInfo (0x04)	11
	10.7 authenticatorClientPIN (0x06)	12
	10.8 AuthenticatorReset (0x07).....	21
11	Message encoding.....	21
	11.1 Commands.....	22
	11.2 Responses	29
	11.3 Error responses	31
12	Interoperating with CTAP1/U2F authenticators.....	33
	12.1 Using the CTAP2 authenticatorMakeCredential command with CTAP1/U2F authenticators	33
	12.2 Using the CTAP2 authenticatorGetAssertion command with CTAP1/U2F authenticators.....	38
13	Transport-specific bindings	41
	13.1 USB	41
	13.2 ISO7816, ISO14443 and near-field communication (NFC).....	50
	Bibliography.....	63

Recommendation ITU-T X.1278

Client to authenticator protocol/Universal 2-factor framework

1 Scope

The application layer protocol in this Recommendation defines the requirements for 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.

2 References

The following ITU-T Recommendations and other references contain provisions which, through reference in this text, constitute provisions of this Recommendation. At the time of publication, the editions indicated were valid. All Recommendations and other references are subject to revision; users of this Recommendation are therefore encouraged to investigate the possibility of applying the most recent edition of the Recommendations and other references listed below. A list of the currently valid ITU-T Recommendations is regularly published. The reference to a document within this Recommendation does not give it, as a stand-alone document, the status of a Recommendation.

- [IETF RFC 2119] IETF RFC 2119 (1997), *Key words for use in RFCs to Indicate Requirement Levels*. <<https://tools.ietf.org/html/rfc2119>>
- [IETF RFC 5480] IETF RFC 5480 (2009), *Elliptic Curve Cryptography Subject Public Key Information*. <<https://tools.ietf.org/html/rfc5480>>
- [IETF RFC 7049] IETF RFC 7049 (2013), *Concise Binary Object Representation (CBOR)*. <<https://tools.ietf.org/html/rfc7049>>
- [U2FRawMsgs] Balfanz, D. *et al.*, (2017), *FIDO U2F Raw Message Formats v1.0*. <<https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-raw-message-formats-v1.2-ps-20170411.html>>
- [WebAuthN] Dirk Balfanz; Alexei Czeskis; Jeff Hodges; J.C. Jones; Michael Jones; Akshay Kumar; Huakai Liao; Rolf Lindemann; Emil Lundberg (2018), *Web Authentication: An API for accessing Public Key Credentials Level 1*. 20 March. W3C Candidate Recommendation. <<https://www.w3.org/TR/webauthn/>>

3 Definitions

None.

4 Abbreviations and acronyms

This Recommendation uses the following abbreviations and acronyms:

AAGUID	Authenticator Attestation Globally Unique Identifier
AES	Advanced Encryption Standard
APDU	Application Protocol Data Unit
API	Application Programming Interface
BLE	Bluetooth Low Energy
CBOR	Concise Binary Object Representation
CID	Channel Identifier

COSE	CBOR Object Signing and Encryption
CTAP	Client to Authenticator Protocol
ECDH	Elliptic Curve Diffie-Hellman
GATT	Generic Attribute profile
HID	Human Interface Device
HMAC	Hash-based Message Authentication Code
JSON	JavaScript Object Notation
LED	Light Emitting Diode
LTK	Long-Term link Key
MTU	Maximum Transmission Unit
NFC	Near-Field Communication
PKCS	Public Key Cryptography Standards
RP	Relying Party
RPA	Resolvable Private Address
U2F	Universal Two Factor
RPID	Relying Party Identity
USB	Universal Serial Bus
UUID	Universally Unique Identifier
UX	User Experience

5 Conventions

5.1 Conformance

All authoring guidelines, diagrams, examples, and notes in this Recommendation are non-normative. Everything else in this Recommendation is normative.

The key words must, must not, required, should, should not, recommended, may and optional in this Recommendation are to be interpreted as described in [IETF RFC 2119].

6 Overview

The client to authenticator protocol is intended to be used in scenarios where a user interacts with a relying party (a website or native app) on a platform (e.g., a PC) which prompts the user to interact with an external authenticator (e.g., a smartphone).

In order to provide evidence of user interaction, an external authenticator implementing this protocol is expected to have a mechanism to obtain a user gesture. Possible examples of user gestures include, a consent button, password, PIN, biometric or a combination of these.

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

7 Relationship to other specifications

This Recommendation is part of the FIDO2 project which includes this client to authenticator protocol (CTAP) and the [b-FIDOSrvrGuid] specifications, and is related to the W3C [WebAuthN] specification. This Recommendation 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 Recommendation. Authenticators implementing CTAP1/U2F are typically referred to as U2F authenticators or CTAP1 authenticators.
- 2) The CTAP2 protocol, whose messages are encoded in the CTAP2 canonical CBOR encoding form. Authenticators implementing CTAP2 are referred to as CTAP2 authenticators, FIDO2 authenticators, or WebAuthn authenticators.

Both CTAP1 and CTAP2 share the same underlying transport, that is, the universal serial bus (USB) human interface device (HID), near-field communication (NFC), and Bluetooth smart/Bluetooth low energy technology (BLE).

The [b-U2FusbHid], [b-U2FNfc], [b-U2FBle] and [U2FRawMsgs] specifications, specifically, are superseded by this Recommendation.

Occasionally, the term "CTAP" may be used without clarifying whether it is referring to CTAP1 or CTAP2. In such cases, it should be understood as referring to the entirety of this Recommendation or portions of this Recommendation 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 in clause 12.

8 Protocol structure

This protocol is specified in three parts as follows.

- **Authenticator API:** At this level of abstraction, each authenticator operation is defined similarly to an application programming interface (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 external authenticators over specific transport (e.g., USB, NFC, Bluetooth). For each transport technology, message bindings are specified for this protocol.

This Recommendation specifies all three of the above parts for external FIDO2 authenticators.

9 Protocol overview

The general protocol between a platform and an authenticator is as follows:

- 1) The platform establishes the connection with the authenticator.
- 2) The platform gets information about the authenticator using the `authenticatorGetInfo` command which helps it determine the capabilities of the authenticator.

- 3) The platform sends a command for an operation if the authenticator is capable of supporting it.
- 4) The authenticator replies with response data or error.

10 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.

10.1 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 (Table 1), which explicitly correspond to those defined in the authenticatorMakeCredential operation section of the web authentication specification:

Table 1 – authenticatorMakeCredential input parameters

Parameter name	Data type	Required?	Definition
clientDataHash	Byte array	Required	Hash of the ClientData contextual binding specified by host. See [WebAuthN].
rp	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, (optionally) a human-friendly RP name, and (optionally) a serialized URL pointing to an 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.
user	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, (optionally) a user name, (optionally) a user display name, and (optionally) a URL pointing

Table 1 – authenticatorMakeCredential input parameters

Parameter name	Data type	Required?	Definition
			to an 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).
pubKeyCredParams	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 [b-IANA-COSE]. This sequence is ordered from most preferred (by the RP) to least preferred.
excludeList	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	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	Sequence of authenticator options	Optional	Parameters to influence authenticator operation, as specified in the Table 2.
pinAuth	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	Unsigned integer	Optional	PIN protocol version chosen by the client

The values in Table 2 are defined for use in the options parameter. All options are Booleans.

Table 2 – authenticatorMakeCredential options parameter values

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.

When such a 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, terminate this procedure and return error code CTAP2_ERR_CREDENTIAL_EXCLUDED.
- 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 options and if any of the requested options cannot be satisfied, terminate this procedure and return the CTAP2_ERR_OPTION_NOT_SUPPORTED error.
- 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 the pinAuth parameter is present and pinProtocol is 1, verify it by matching it against the 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.

If the pinAuth parameter is not present and clientPin is set on the authenticator, return CTAP2_ERR_PIN_REQUIRED error.

- 6) 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 light emitting diode (LED) light). Request permission to create a credential. If the user declines permission, return the CTAP2_ERR_OPERATION_DENIED error.
- 7) Generate a new credential key pair for the algorithm specified.
- 8) If "rk" in the options parameter is set to true, the following should be carried out:
 - 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 the authenticator does not have enough internal storage to persist the new credential, return CTAP2_ERR_KEY_STORE_FULL.
- 9) Generate an attestation statement for the newly-created key using clientDataHash.

Upon success, the authenticator returns an attestation object (see Table 3) in its response as defined in [WebAuthN]:

Table 3 – Attestation object

Member name	Data type	Required?	Definition
authData	Sequence of bytes	Required	The authenticator data object.
fmt	String	Required	The attestation statement format identifier.
attStmt	Sequence of bytes, the structure of which depends on the attestation statement format identifier	Required	The attestation statement, whose format is identified by the "fmt" object member. The client treats it as an opaque object.

10.2 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 as shown in Table 4, which explicitly correspond to those defined in the authenticatorGetAssertion operation section of the Web Authentication specification [WebAuthN]:

Table 4 – authenticatorGetAssertion input parameters

Parameter name	Data type	Required?	Definition
rpId	String	Required	Relying party identifier. See [WebAuthN].
clientDataHash	Byte array	Required	Hash of the serialized client data collected by the host. See [WebAuthN].
allowList	Sequence of PublicKeyCredentialDescriptors	Optional	A sequence of PublicKeyCredentialDescriptor structures, each denoting a credential, as specified in [WebAuthN]. The authenticator is requested to only generate an assertion using one of the denoted credentials.
extensions	CBOR map of extension identifier → authenticator extension input values	Optional	Parameters to influence authenticator operation. These parameters might be authenticator specific.
options	Sequence of authenticator options	Optional	Parameters to influence authenticator operation, as specified in Table 5.

Table 4 – authenticatorGetAssertion input parameters

Parameter name	Data type	Required?	Definition
pinAuth	Byte array	Optional	The first 16 bytes of HMAC-SHA-256 of clientDataHash using pinToken which the platform obtained from the authenticator: <code>HMAC-SHA-256(pinToken, clientDataHash)</code> .
pinProtocol	Unsigned integer	Optional	PIN protocol version selected by the client.

The values in Table 5 are defined for use in the options parameter (All options are Booleans).

Table 5 – authenticatorGetAssertion options parameter values

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 identifies the user in order to complete the request. Examples of such gestures are fingerprint scan or a PIN.

When such a 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.
- 2) If pinAuth parameter is present and pinProtocol is 1, verify it by matching it against the 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.

If pinAuth parameter is not present and clientPin has been set on the authenticator, set the "uv" bit to 0 in the response.
- 3) 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.
- 4) 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 the device does not support user-identifiable gestures, return the CTAP2_ERR_OPTION_NOT_SUPPORTED 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.
- 5) If no credentials were located in step 1, return CTAP2_ERR_NO_CREDENTIALS.
 - 6) If only one credential was located in step 1, go to step 9.
 - 7) Order the credentials by the time they were created. The first credential is the most recent credential that was created.
 - 8) If the authenticator does not have a display:
 - Remember the authenticatorGetAssertion parameters.
 - Create a counter and set it to the total number of credentials.
 - Start a timer. This is used during authenticatorGetNextAssertion command.
 - Update the response to include the first credential's publicKeyCredentialUserEntity information and numberOfCredentials.
 - 9) If the 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.
 - 10) Sign the clientDataHash along with authData with the selected credential, using the structure specified in [WebAuthN].

Upon success, the authenticator returns the structure outlined in Table 6 in its response:

Table 6 – authenticator response structure

Member name	Data type	Required?	Definition
credential	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	Byte array	Required	The signed-over contextual bindings made by the authenticator, as specified in [WebAuthN].
signature	Byte array	Required	The assertion signature produced by the authenticator, as specified in [WebAuthN].

Table 6 – authenticator response structure

Member name	Data type	Required?	Definition
user	PublicKeyCredentialUserEntity	Required	PublicKeyCredentialUserEntity structure containing the user account information. 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. The 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	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.

10.3 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 always follows a call to authenticatorGetAssertion or authenticatorGetNextAssertion.

When such a request is received, the authenticator performs the following procedure:

- 1) If the authenticator does not remember any authenticatorGetAssertion parameters, return CTAP2_ERR_NOT_ALLOWED.
- 2) If the credential counter is 0, 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.
- 4) Sign the clientDataHash with the credential using the credential counter as index (e.g., credentials[n] assuming 1-based array), using the structure specified in [WebAuthN].
- 5) Reset the timer.
- 6) Decrement the credential counter.

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

10.4 Client logic

If the 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 user experience (UX) that displays credentialInfo list.
- 3) Let the user select which credential to use.
- 4) Return the value of the 'response' member associated with the user choice.
- 5) Discard all other responses.

10.5 authenticatorCancel (0x03)

Using this method, the host can request the authenticator to cancel all ongoing operations and return to a ready state. It takes no input parameters and returns success or failure.

10.6 authenticatorGetInfo (0x04)

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

Upon success, the authenticator returns the structure outlined in Table 7.

Table 7 – authenticatorGetInfo return structure

Member name	Data type	Required?	Definition
versions	Sequence of strings	Required	List of supported versions
extensions	Sequence of strings	Optional	List of supported extensions
aaguid	Byte string	Required	The claimed AAGUID. 16 bytes in length and encoded the same as MakeCredential AuthenticatorData, as specified in [WebAuthN].
options	Map	Optional	List of supported options
maxMsgSize	Unsigned integer	Optional	Maximum message size supported by the authenticator
pinProtocols	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 as outlined in Table 8 with the following list of supported options:

Table 8 – authenticatorGetInfo supported options

Option ID	Definition	Default
plat	Platform device: indicates that the device is attached to the client and therefore cannot 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 yet been set. If absent, it indicates that the device is not capable of accepting a PIN from the client.	Not supported
up	User presence: indicates that the device is capable of testing user presence as part of the authenticatorGetAssertion request.	true
uv	User verification: indicates that the device is capable of verifying the user as part of the authenticatorGetAssertion request.	false

10.7 authenticatorClientPIN (0x06)

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

This command is used by the platform to establish key agreement with the 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 input parameters outlined in Table 9:

Table 9 – authenticatorClientPIN input parameters

Parameter name	Data type	Required?	Definition
pinProtocol	Integer	Required	PIN protocol version chosen by the client. For this version of the spec, this shall be the number 1.
subCommand	Integer	Required	The authenticator client PIN sub-command currently being requested
keyAgreement	COSE_KEY	Optional	Public key of platformKeyAgreementKey
pinAuth	Byte array	Optional	The 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	Byte array	Optional	Encrypted new PIN using sharedSecret. Encryption is done over UTF-8 representation of new PIN.
pinHashEnc	Byte array	Optional	Encrypted first 16 bytes of SHA-256 of PIN using sharedSecret.

Table 9 – authenticatorClientPIN input parameters

Parameter name	Data type	Required?	Definition
getKeyAgreement	Boolean	Optional	Asks authenticator to return public key of its authenticatorKeyAgreementKey for getting SharedSecret from the authenticator.
getRetries	Boolean	Optional	Asks authenticator to return number of PIN attempts remaining before lockout.

The list of sub commands for PIN Protocol Version 1 are given in Table 10.

Table 10 – sub commands for PIN Protocol Version 1

Sub-command name	Sub-command number
Get Retries	1
Get Key Agreement	2
Set PIN	3
Change PIN	4
Get PIN token	5

Upon success, the authenticator returns the following structure in its response (see Table 11).

Table 11 – authenticatorClientPIN response structure

Parameter name	Data type	Required?	Definition
KeyAgreement	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.
pinToken	Byte array	Optional	Encrypted pinToken using sharedSecret to be used in subsequent authenticatorMakeCredential and authenticatorGetAssertion operations.
retries	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.

10.7.1 Client PIN support requirements

- The platform has to fulfil the 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 255 bytes
- The authenticator has to fulfil the following PIN support requirements:
 - Minimum PIN Length: 4 bytes
 - Maximum PIN Length: 255 bytes
 - Maximum incorrect PIN retry count: 8
 - Each correct PIN entry resets retries counter

- Once the authenticator reaches the maximum incorrect PIN retry count, the authenticator has to be reset before any further operations requiring the use of the 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 characters.

10.7.2 Authenticator configuration operations upon power up

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

Below are the operations the authenticator performs on each power up:

- 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 [b-RFC 6090] Section 4.1 and [b-SP800-56A] for more Elliptic Curve Diffie-Hellman (ECDH) key agreement protocol details.
- Generate **"pinToken"**:
 - Generate a random integer of the length, which is a multiple of 16 bytes (AES block length).
 - "pinToken" is used so that there is minimum burden on the authenticator and the platform does not have to send actual encrypted PIN to the authenticator in normal authenticator usage scenarios. This also provides more security as the actual PIN is not being sent, not 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.

10.7.3 Getting sharedSecret from the authenticator

The platform sets the ECDH key agreement to arrive at sharedSecret to be used only during that transaction. The authenticator does not have to keep a list of sharedSecrets for all active sessions. If there are subsequent authenticatorClientPIN transactions, a new sharedSecret is generated each time.

The platform performs the following operations to arrive at the sharedSecret:

- The platform sends authenticatorClientPIN command by setting getKeyAgreement parameter to true.
- The platform optionally can set getRetries parameter to true to get the retries count. Retries count is the number of attempts remaining before lockout so when the device is near the authenticator lockout stage, the platform can optionally warn the user to be careful while entering PIN.
- The authenticator replies with the public key of authenticatorKeyAgreementKey, "*aG*".
 - The authenticator optionally also sends retries count if getRetries parameter is set to true.
- The platform generates **"platformKeyAgreementKey"**:
 - The 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.
- The platform generates **"sharedSecret"**
 - The platform generates "sharedSecret" using SHA-256 over ECDH key agreement protocol using private key of platformKeyAgreementKey, "*b*" and public key of authenticatorKeyAgreementKey, "*aG*": $SHA-256((b \cdot aG) \cdot x)$.

- SHA-256 is done over only "x" curve point of baG.
- See [b-RFC 6090] Section 4.1 and appendix (C.2) of [b-SP800-56A] for more ECDH key agreement protocol details and key representation.

10.7.4 Setting a new PIN

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

- The platform gets sharedSecret from the authenticator.
- The platform collects new PIN ("newPinUnicode") from the user in Unicode format.
 - The 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".
 - The platform checks the byte length of "newPin" against the max UTF-8 representation limit of 255 bytes and returns CTAP2_ERR_PIN_POLICY_VIOLATION if the check fails.
- The platform sends authenticatorClientPIN command with the following parameters to the authenticator:
 - 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 a minimum of 64 bytes length. This is to prevent a 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.
- The authenticator performs the following operations upon receiving the request:
 - 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 only over the "x" curve point of "abG"
 - See [b-RFC 6090] section 4.1 and Appendix C.2 of [b-SP800-56A] for more ECDH key agreement protocol details and key representation.
 - The 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.
 - The authenticator decrypts newPinEnc using above "sharedSecret" producing newPin and checks newPin length against a minimum PIN length of 4 characters.
 - 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, the authenticator returns CTAP2_ERR_PIN_POLICY_VIOLATION error.
 - The authenticator may have additional constraints for the PIN policy. The current specification only enforces a minimum length of 4 characters.

- The authenticator stores `LEFT(SHA-256(newPin), 16)` on the device and returns CTAP2_OK.

10.7.5 Changing existing PIN

The following operations are performed to change an existing PIN:

- The platform gets sharedSecret from the authenticator.
- The platform collects current PIN ("curPinUnicode") and new PIN ("newPinUnicode") from the user.
 - The 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"
 - The platform checks the byte length of "curPin" and "newPin" against the max UTF-8 representation limit of 255 bytes and returns CTAP2_ERR_PIN_POLICY_VIOLATION if the check fails.
- The platform sends authenticatorClientPIN command with the following parameters to the authenticator:
- 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 a minimum of 64 bytes length. This is to prevent a 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.
- The authenticator performs the following operations upon receiving the request:
 - 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 only over the "x" curve point of "abG"
 - See [b-RFC 6090] section 4.1 and Appendix C.2 of [b-SP800-56A] for more ECDH key agreement protocol details and key representation.
 - The 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.
 - The authenticator decrypts pinHashEnc and verifies against its internal stored `LEFT(SHA-256(curPin), 16)`.
 - If a mismatch is detected the authenticator generate new "authenticatorKeyAgreementKey" first and then returns CTAP2_ERR_PIN_INVALID error.

- 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 [b-RFC 6090] section 4.1 and [b-SP800-56A] for more ECDH key agreement protocol details.
- The authenticator decrypts newPinEnc using the above "sharedSecret" producing newPin and checks newPin length against a minimum PIN length of 4 characters.
 - 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 the minimum PIN length check fails, the authenticator returns CTAP2_ERR_PIN_POLICY_VIOLATION error.
 - The authenticator may have additional constraints for PIN policy. The current specification only enforces a minimum length of 4 characters.
- The authenticator stores `LEFT(SHA-256(newPin), 16)` on the device and returns CTAP2_OK.

10.7.6 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 high security without any additional roundtrips every time (except for the first key-agreement phase) and its overhead is minimal.

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

- The platform gets sharedSecret from the authenticator.
- The platform collects PIN from the user.
- The platform sends authenticatorClientPIN command with the following parameters to the authenticator:
 - keyAgreement: public key of platformKeyAgreementKey, "bG".
 - pinHashEnc: `AES256-CBC(sharedSecret, IV=0, LEFT(SHA-256(PIN), 16))`.
- The authenticator performs the following operations upon receiving the request:
 - 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 "x" curve point of "abG"
 - See [b-RFC 6090] Section 4.1 and appendix (C.2) of [b-SP800-56A] for more ECDH key agreement protocol details and key representation.
 - The authenticator decrypts pinHashEnc and verifies against its internal stored `LEFT(SHA-256(curPin), 16)`.
 - If a mismatch is detected the authenticator generate new "authenticatorKeyAgreementKey" first and then returns CTAP2_ERR_PIN_INVALID error.
 - 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 [b-RFC 6090] section 4.1 and [b-SP800-56A] for more ECDH key agreement protocol details.

- The authenticator returns the 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.

10.7.7 Using pinToken

The 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. The authenticator also can expire the pinToken based on certain conditions like changing a PIN, timeout happening on the authenticator, machine waking up from a 'suspend' state, etc. If the pinToken has expired, the authenticator will return CTAP2_ERR_PIN_TOKEN_EXPIRED and the platform can act on the error accordingly.

10.7.7.1 Using pinToken in authenticatorMakeCredential

The following operations are performed to use pinToken in authenticatorMakeCredential API:

- The platform gets pinToken from the authenticator.
- The platform sends authenticatorMakeCredential command with the following additional optional parameter:
 - pinAuth: `LEFT(HMAC-SHA-256(pinToken, clientDataHash), 16)`.
 - The platform sends the first 16 bytes of the HMAC-SHA-256 result.
- The authenticator verifies pinAuth by generating `LEFT(HMAC-SHA-256(pinToken, clientDataHash), 16)` and matching against input pinAuth parameter.
- The authenticator returns authenticatorMakeCredential response with "uv" bit set to 1.

If platform sends zero length pinAuth, the 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.

10.7.7.2 Using pinToken in authenticatorGetAssertion

The following operations are performed to use pinToken in authenticatorGetAssertion API:

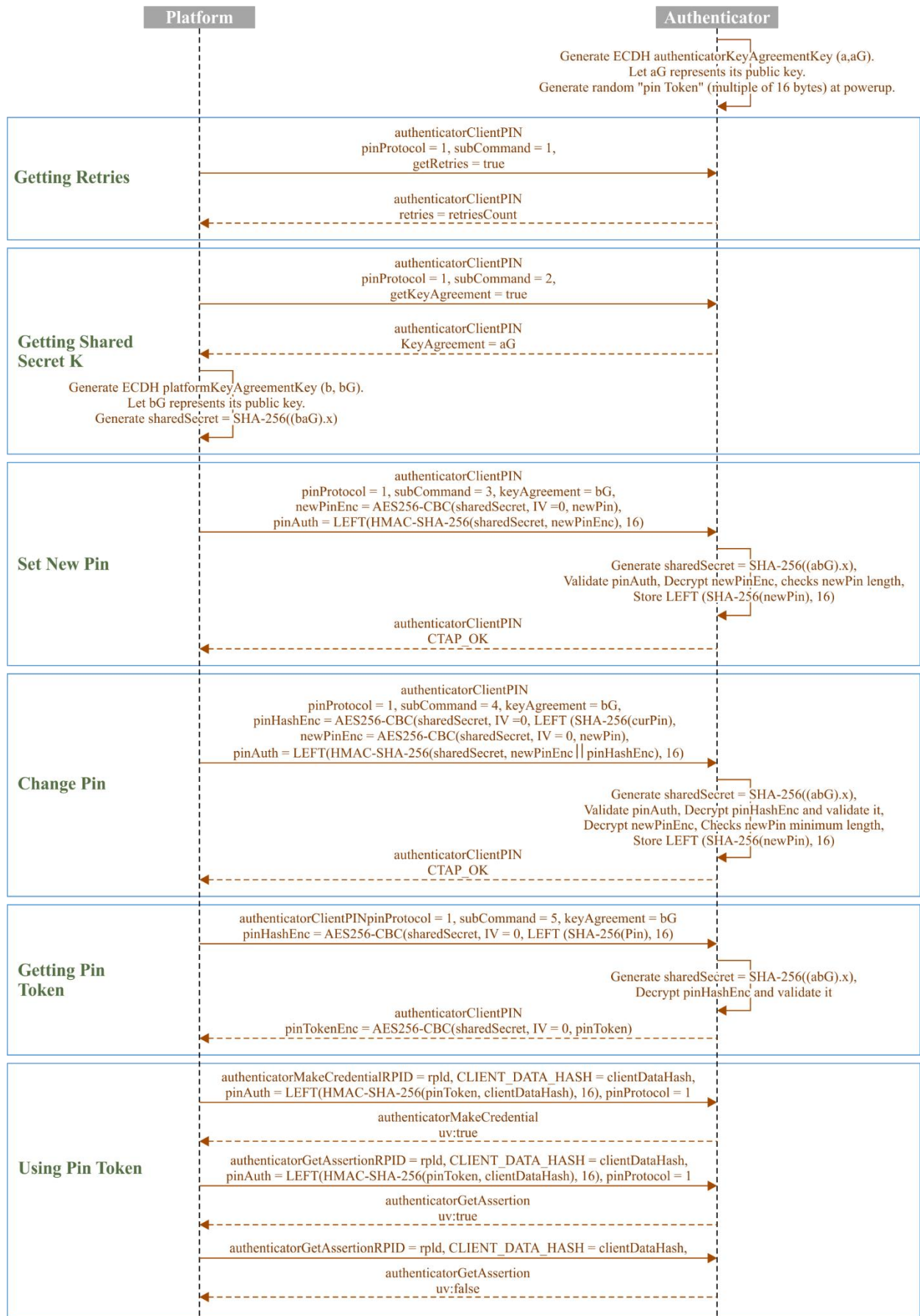
- The platform gets pinToken from the authenticator.
- The platform sends authenticatorGetAssertion command with the following additional optional parameter:
 - pinAuth: `LEFT(HMAC-SHA-256(pinToken, clientDataHash), 16)`.
- The authenticator verifies pinAuth by generating `LEFT(HMAC-SHA-256(pinToken, clientDataHash), 16)` and matching against input pinAuth parameter.
- The authenticator returns authenticatorGetAssertion response with "uv" bit set to 1.

If the platform sends zero length pinAuth, the 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.

10.7.7.3 Without pinToken in authenticatorGetAssertion

The following operations are performed without using pinToken in authenticatorGetAssertion API:

- The platform sends authenticatorGetAssertion command without pinAuth optional parameter.
- The authenticator returns authenticatorGetAssertion response with "uv" bit set to 0.



X.1278(18)_F01

Figure 1 – Client pin

10.8 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 is outside the scope of this Recommendation.

11 Message encoding

Many transport types (e.g., Bluetooth Smart) are bandwidth constrained, and serialization formats such as JavaScript object notation (JSON) are too heavyweight for such environments. For this reason, all encoding is done using the concise binary encoding CBOR [IETF RFC 7049].

To reduce the complexity of the messages and the resources required to parse and validate them, all messages must use canonical CBOR as specified below. All encoders must generate canonical CBOR without duplicate map keys. All decoders should enforce canonical CBOR and should reject messages with duplicate map keys. Canonical CBOR for CTAP (CTAP1/U2F and CTAP2) 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 expression of lengths in major types 2 to 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 from the lowest value to highest. Sorting is performed on the bytes of the representation of the key data items without paying attention to the 3/5 bit splitting for major types. 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.

As 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.

Similarly, 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 there are map keys 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 the authenticator are called "commands" and messages from the authenticator to the host are called "replies". All values are big-endian encoded.

11.1 Commands

All commands are structured as shown in Table 12.

Table 12 – authenticatorReset commands

Name	Length	Required?	Definition
Command value	1 byte	Required	The value of the command to execute
Command parameters	variable	Optional	CBOR [IETF RFC 7049] encoded set of parameters. Some commands have parameters, while others do not (see below)

The assigned values for commands and their descriptions are shown in Table 13.

Table 13 – authenticatorReset assigned values

Command name	Command value	Has parameters?
authenticatorMakeCredential	0x01	yes
authenticatorGetAssertion	0x02	yes
authenticatorCancel	0x03	no
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 [IETF RFC 7049]. 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.charCodeAtAt(0)),  
  icon: "https://pics.acme.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"
7828		# text(40)
	68747470733a2f2f706963732e61636d	# "https://pics.acme.com/00/p/aBjjjpqPb.png"
	652e636f6d2f30302f702f61426a6a6a	# ...
	707150622e706e67	# ...
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 in Table 14.

Table 14 – parameter map key and value types

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).

Table 14 – parameter map key and value types

Command	Parameter name	Key	Value type
	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 specification 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).
	getKeyAgreement	0x07	Boolean. (CBOR major type 7, additional simple value information 20(False)/21(True)).
	getRetries	0x08	Boolean. (CBOR major type 7, additional simple value information 20(False)/21(True)).

EXAMPLE 4

The following is a complete encoding example of the `authenticatorMakeCredential` command (using the 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'687134968222ec17202e42505f8ed2b16ae22f16bb05b88c25db9e602645f141'
    6ae22f16bb05b88c25db9e602645f141                                     #
  02                                     # unsigned(2) - rp
  a2                                     # map(2)
    62                                     # text(2)
      6964                             # "id"
    68                                     # text(8)

```

```

        61636d652e636f6d      # "acme.com"
64      6e616d65              # text(4)
        6e616d65              # "name"
64      41636d65              # 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      28                    # text(40)
        68747470733a2f2f706963732e616 # "https://pics.acme.com/00/p/aBjjjpqPb.png"
        36d652e636f6d2f30302f702f6142 #
        6a6a6a707150622e706e67      #
64      # text(4)
        6e616d65              # "name"
76      # text(22)
        6a6f686e70736d697468406578616 # "johnpsmith@example.com"
        d706c652e636f6d          #
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)
70      # text(16)
        6b657953746f72616765446576696 # "keyStorageDevice"
        365                    #
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)
        3045022013f73c5d9d530e8cc15cc       # signature...
        9bd96ad586d393664e462d5f05612       # ...
        35e6350f2b728902210090357ff91       # ...
        0ccb56ac5b596511948581c8fddb4       # ...
        a2b79959948078b09f4bdc6229         # ...
63                                           # text(3)
        783563                                # "x5c"
81                                           # array(1)
59 0197                                       # bytes(407)
        3082019330820138a003020102         # certificate...
        020900859b726cb24b4c29300a         # ...
        06082a8648ce3d040302304731         # ...
        0b300906035504061302555331         # ...
        143012060355040a0c0b597562         # ...
        69636f20546573743122302006         # ...
        0355040b0c1941757468656e74         # ...
        696361746f7220417474657374         # ...
        6174696f6e301e170d31363132         # ...
        30343131353530305a170d3236         # ...
        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)
68                                           # text(8)
        61636d652e636f6d                   # "acme.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)
03030303030303030303030303030303 # credential ID
03030303030303030303030303030303 # ...
03030303030303030303030303030303 # ...
030303030303030303030303 # ...
64 # text(4)
74797065 # "type"
6a # text(10)
7075626C69632D6B6579 # "public-key"
05 # unsigned(5)
a1 # map(1)
62 # text(2)
747569 # "uv"
f5 # true

```

authenticatorGetAssertion_Response response:

```

00 # status = success
a3 # map(5)
01 # unsigned(1) - Credential
a2 # map(2)
62 # text(2)
6964 # "id"
58 40 # bytes(64)
f22006de4f905af68a43942f024f2 # credentialId
a5ece603d9c6d4b3df8be08ed01fc # ...
442646d034858ac75bed3fd580bf9 # ...
808d94fcbee82b9b2ef6677af0adc # ...
c35852ea6b9e # ...
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)
6b # text(11)
646973706c61794e616d65 # "displayName"
6d # text(13)
4a6f686e20502e20536d697468 # "John P. Smith"
64 # text(4)
6e616d65 # "name"
76 # text(22)
6a6f686e70736d697468406578616d # "johnpsmith@example.com"
706c652e636f6d # ...

```

```

62                                     # text(2)
    6964                             # "id"
58 20                               # bytes(32)
    3082019330820138a003020102      # userid
    3082019330820138a003020102      # ...
    308201933082                     # ...
64                                   # text(4)
    69636f6e                         # "icon"
7828                                # text(40)
    68747470733a2f2f706963732e6163  # "https://pics.acme.com/00/p/aBjjjpqPb.png"
    6d652e636f6d2f30302f702f61426a  # ...
    6a6a707150622e706e67            # ...
05                                   # unsigned(5) - numberOfCredentials
01                                   # unsigned(1)

```

11.2 Responses

All responses are structured as shown in Table 15:

Table 15 – Response structure

Name	Length	Required?	Definition
Status	1 byte	Required	The status of the response. 0x00 means success; all other values are errors. See Table 17 for error 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 in Table 16.

Table 16 – Map key and value types

Response message	Member name	Key	Value type
authenticatorMakeCredential_Response	fmt	0x01	text string (CBOR major type 3).
	authData	0x02	byte string (CBOR major 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).

Table 16 – Map key and value types

Response message	Member name	Key	Value type
	publicKeyCredentialUserEntity	0x04	definite length map (CBOR major type 5). Must not be present if UV bit is not set.
	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].

Table 16 – Map key and value types

Response message	Member name	Key	Value type
	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	CBOR definite length array (CBOR major type 4) of CBOR unsigned integers (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.
	pinToken	0x02	byte string (CBOR major type 2).
	retries	0x03	Unsigned integer (CBOR major type 0). This is number of retries left before lockout.

11.3 Error responses

The error response values range from 0x01 – 0xff. This range is split based on the error type (see Table 17).

Error response values in the range between CTAP2_OK and CTAP2_ERR_SPEC_LAST are reserved for specification 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 (Table 17: Error responses) 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 extensions.

Table 17 – Error responses

Code	Name	Description
0x00	CTAP1_ERR_SUCCESS	Indicates successful response.
0x01	CTAP1_ERR_INVALID_COMMAND	The command is not a valid CTAP (CTAP1/U2F and CTAP2) 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.
0x10	CTAP2_ERR_CBOR_PARSING	Error while parsing CBOR.
0x11	CTAP2_ERR_CBOR_UNEXPECTED_TYPE	Invalid/unexpected CBOR error.
0x12	CTAP2_ERR_INVALID_CBOR	Error when parsing CBOR.
0x13	CTAP2_ERR_INVALID_CBOR_TYPE	Invalid or unexpected CBOR type.
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.
0x17	CTAP2_ERR_TOO_MANY_ELEMENTS	Limit for number of items exceeded.
0x18	CTAP2_ERR_EXTENSION_NOT_SUPPORTED	Unsupported extension.
0x19	CTAP2_ERR_CREDENTIAL_EXCLUDED	Valid credential found in the excludeList.
0x20	CTAP2_ERR_CREDENTIAL_NOT_VALID	Credential not valid for authenticator.
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	The 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.
0x29	CTAP2_ERR_NOT_BUSY	The authenticator cannot cancel as it is not busy.
0x2A	CTAP2_ERR_NO_OPERATION_PENDING	No outstanding operations.

Table 17 – Error responses

Code	Name	Description
0x2B	CTAP2_ERR_UNSUPPORTED_OPTION	Unsupported option.
0x2C	CTAP2_ERR_INVALID_OPTION	Unsupported option.
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 blocked.
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	The authenticator cannot handle this request due to memory constraints.
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.

12 Interoperating with CTAP1/U2F authenticators

This clause 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 fulfil. The processes for RPs to use to verify CTAP1/U2F based authenticatorMakeCredential and authenticatorGetAssertion responses are also defined below. The platform may choose to skip this feature and work only with CTAP (CTAP1/U2F and CTAP2) devices.

12.1 Using the CTAP2 authenticatorMakeCredential command with CTAP1/U2F authenticators

The platform follows the following procedure (see also Figure 2).

- 1) The platform tries to get information about the authenticator by sending authenticatorGetInfo command as specified in CTAP2 protocol overview.

- 2) CTAP1/U2F authenticator returns a command error or improperly formatted CBOR response. For any failure the platform may fall back to CTAP1/U2F protocol.
- 3) Map CTAP2 authenticatorMakeCredential request to U2F_REGISTER request.
- 4) The platform verifies that CTAP2 request does not have any parameters that CTAP1/U2F authenticators cannot fulfil.
 - All of the conditions below must be true for the platform to proceed to the next step. If any of the conditions below is not true, platform errors out with CTAP2_ERR_OPTION_NOT_SUPPORTED.
 - 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 indicates that 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).
- 5) Send the U2F_REGISTER request to the authenticator as specified in [U2FRawMsgs].
- 6) Map the U2F registration response message (see the "Registration Response Message: Success" section of [U2FRawMsgs]) to a CTAP2 authenticatorMakeCredential response message:
 - Generate authenticatorData from the U2F registration response message received from the authenticator:
 - Initialize attestationData:
 - Let credentialIdLength be a 2-byte unsigned big-endian integer representing the 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 ([IETF RFC 5480] clause 2.2) to COSE_Key representation ([b-RFC 8152] section 7).
 - Let attestationData be a byte array with the structure outlined in Table 18.

Table 18 – attestationData 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 <code>credentialIdLength</code> bytes.
<code>credentialIdLength</code>	Credential ID.	Initialized with <code>credentialID</code> bytes.
77	The credential public key.	Initialized with <code>coseEncodedCredentialPublicKey</code> 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 the Authenticator Data section of [WebAuthN].
- Let `signCount` be a 4-byte unsigned integer initialized to zero.
- Let `authenticatorData` be a byte array with the structure outlined in Table 19.

Table 19 – authenticatorData structure

Length (in bytes)	Description	Value
32	SHA-256 hash of the <code>rp.id</code> .	Initialized with <code>rpIdHash</code> bytes.
1	Flags	Initialized with <code>flags</code> ' value.
4	Signature counter (<code>signCount</code>).	Initialized with <code>signCount</code> bytes.
Variable Length	Attestation Data.	Initialized with <code>attestationData</code> '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"'s value 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"'s value to `authenticatorData`.
 - Set "fmt"'s value to "fido-u2f".
 - Set "attStmt"'s value to `attestationStatement`.

7) Return `attestationObject` to the caller.

Example 6

Sample CTAP2 authenticatorMakeCredential request (CBOR):

```
{1: h'687134968222EC17202E42505F8ED2B16AE22F16BB05B88C25DB9E602645F141 ',
 2: {"id": "acme.com",
```

```

    "name": "acme.com"},
3: {"id": "1098237235409872",
    "name": "johnpsmith@example.com",
    "icon": "https://pics.acme.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.

```
687134968222EC17202E42505F8ED2B16AE22F16BB05B8C25DB9E602645F141 # clientdatahash
1194228DA8FDBDEEFD261BD7B6595CFD70A50D70C6407BCF013DE96D4EFB17DE # rpidhash
```

Sample CTAP1/U2F response from the device

[illegible]

Authenticator Data from CTAP1/U2F response

```
1194228DA8FDBDEEFD261BD7B6595CFD70A50D70C6407BCF013DE96D4EFB17DE # rpidhash
41 # flags
00000000 # Sign Count
00000000000000000000000000000000 # AAGUID
0040 # Key Handle Length (1 Byte)
3EBD89BF77EC509755EE9C2635EFAAAC7B2B9C5CEF1736C3717DA48534C8C6B6 # Key Handle (Key Handle Length Bytes)
54D7FF945F50B5CC4E78055BDD396B64F78DA2C5F96200CCD415CD08FE420038 # ...
A5010203262001215820E87625896EE4E46DC032766E8087962F36DF9DFE8B56 # Public Key
7F3763015B1990A60E1422582027DE612D66418BDA1950581EBC5C8C1DAD710C # ...
B14C22F8C97045F4612FB20C91 # ...
```

Mapped CTAP2 authenticatorMakeCredential response(CBOR)

[illegible]

4886F70D01010B050003820101009F9B052248BC4CF42CC5991FCAABAC9B651B
BE5BDCDC8EF0AD2C1C1FFB36D18715D42E78B249224F92C7E6E7A05C49F0E7E4
C881BF2E94F45E4A21833D7456851D0F6C145A29540C874F3092C934B43D222B
8962C0F410CEF1DB75892AF116B44A96F5D35ADEA3822FC7146F6004385BCB69
B65C99E7EB6919786703C0D8CD41E8F75CCA44AA8AB725AD8E799FF3A8696A6F
1B2656E631B1E40183C08FDA53FA4A8F85A05693944AE179A1339D002D15CABD
810090EC722EF5DEF9965A371D415D624B68A2707CAD97BCDD1785AF97E258F3
3DF56A031AA0356D8E8D5EBCADC74E071636C6B110ACE5CC9B90DFEACAE640FF
1BB0F1FE5DB4EFF7A95F060733F5'] } }

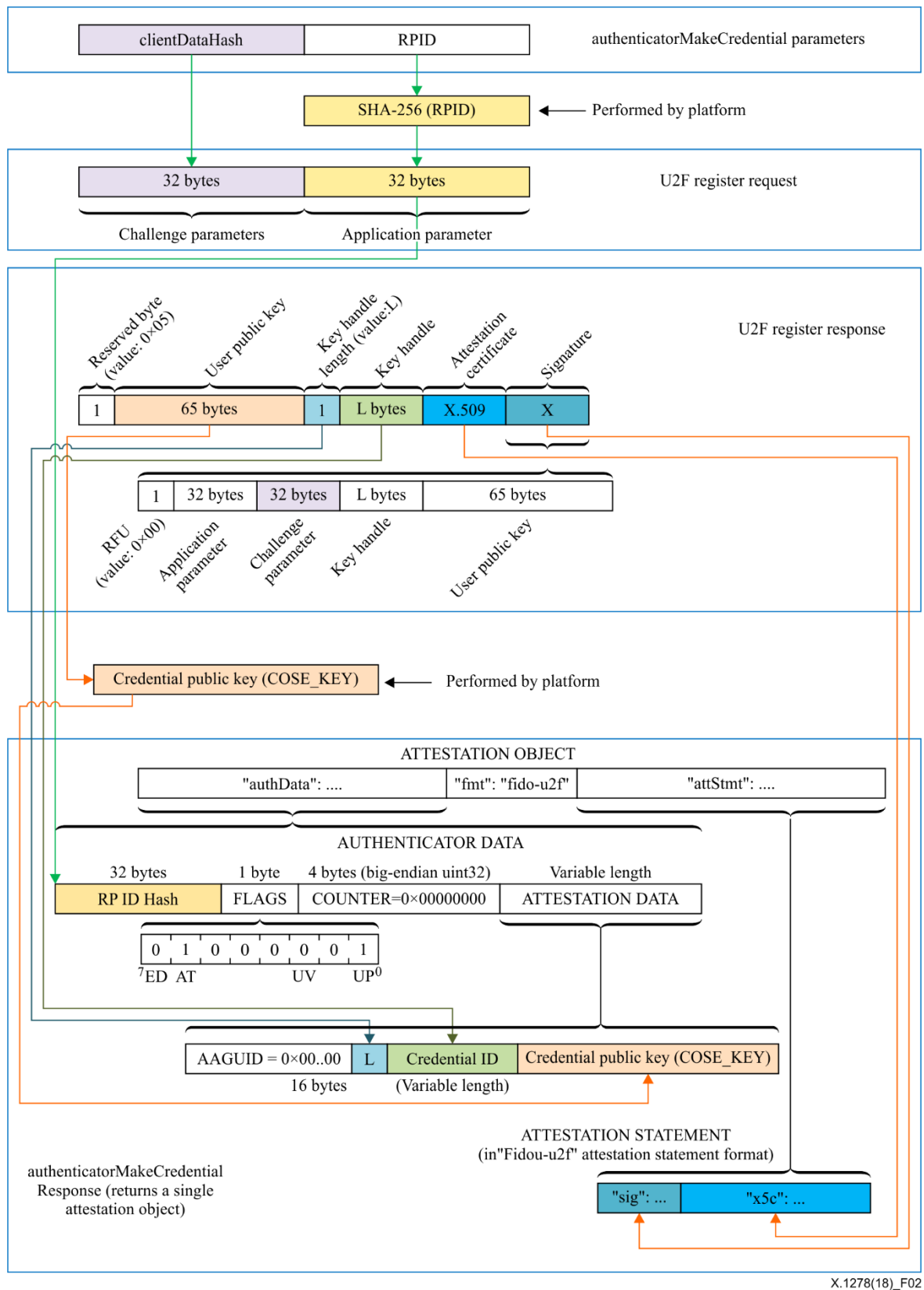


Figure 2 – Mapping: WebAuthn authenticatorMakeCredential to and from CTAP1/U2F registration messages

12.2 Using the CTAP2 authenticatorGetAssertion command with CTAP1/U2F authenticators

The platform follows the following procedure (see also Figure 3).

- 1) The 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:
 - The platform verifies that CTAP2 request does not have any parameters that CTAP1/U2F authenticators cannot fulfil:
 - All of the conditions below must be true for the platform to proceed to the next step. If any of the conditions below is not true, platform errors out with CTAP2_ERR_OPTION_NOT_SUPPORTED.
 - Options must not include "uv" set to true.
 - allowList must have at least one credential.
 - If allowList has more than one credential the platform has to loop over the list and send individual different U2F_AUTHENTICATE commands to the authenticator. For each credential in the credential list, map CTAP2 authenticatorGetAssertion request to U2F_AUTHENTICATE as below:
 - Let `controlByte` be a byte initialized as follows:
 - For USB, set it to 0x07 (check-only). This should prevent a call getting blocked on waiting for user input. If a 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 will not 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` be 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 structure outlined in Table 20.

Table 20 – u2fAuthenticateRequest structure

Length (in bytes)	Description	Value
1	Control byte	Initialized with <code>controlByte</code> 's value.
32	Challenge parameter	Initialized with <code>clientDataHash</code> parameter bytes.
32	Application parameter	Initialized with <code>rpIdHash</code> bytes.
1	Key handle length	Initialized with <code>keyHandleLength</code> 's value.
<code>keyHandleLength</code>	Key handle	Initialized with <code>credentialID</code> bytes.

- 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:
 - Let **flags** be a byte whose zeroth bit (bit 0, UP) is set to 1 if CTAP1/U2F response user presence byte is set to 1, and all other bits are zero (bit zero is the least significant bit). See also the 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 the structure outlined in Table 21.

Table 21 – authenticatorData 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 the signature field from CTAP1/U2F authentication response message.

EXAMPLE 7

Sample CTAP2 authenticatorGetAssertion request (CBOR):

```
{1: "acme.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 Byte)
3EBD89BF77EC509755EE9C2635EFAAAC7B2B9C5CEF1736C3717DA48534C8C6B6 # Key Handle (Key Handle Length Bytes)
54D7FF945F50B5CC4E78055BDD396B64F78DA2C5F96200CCD415CD08FE420038 # ...
```

Sample CTAP1/U2F Response from the device

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

Authenticator Data from CTAP1/U2F Response

```
1194228DA8FDBDEEFD261BD7B6595CFD70A50D70C6407BCF013DE96D4EFB17DE # rpIdhash
01 # User Presence (1 Byte)
0000003B # Sign Count (4 Bytes)
```

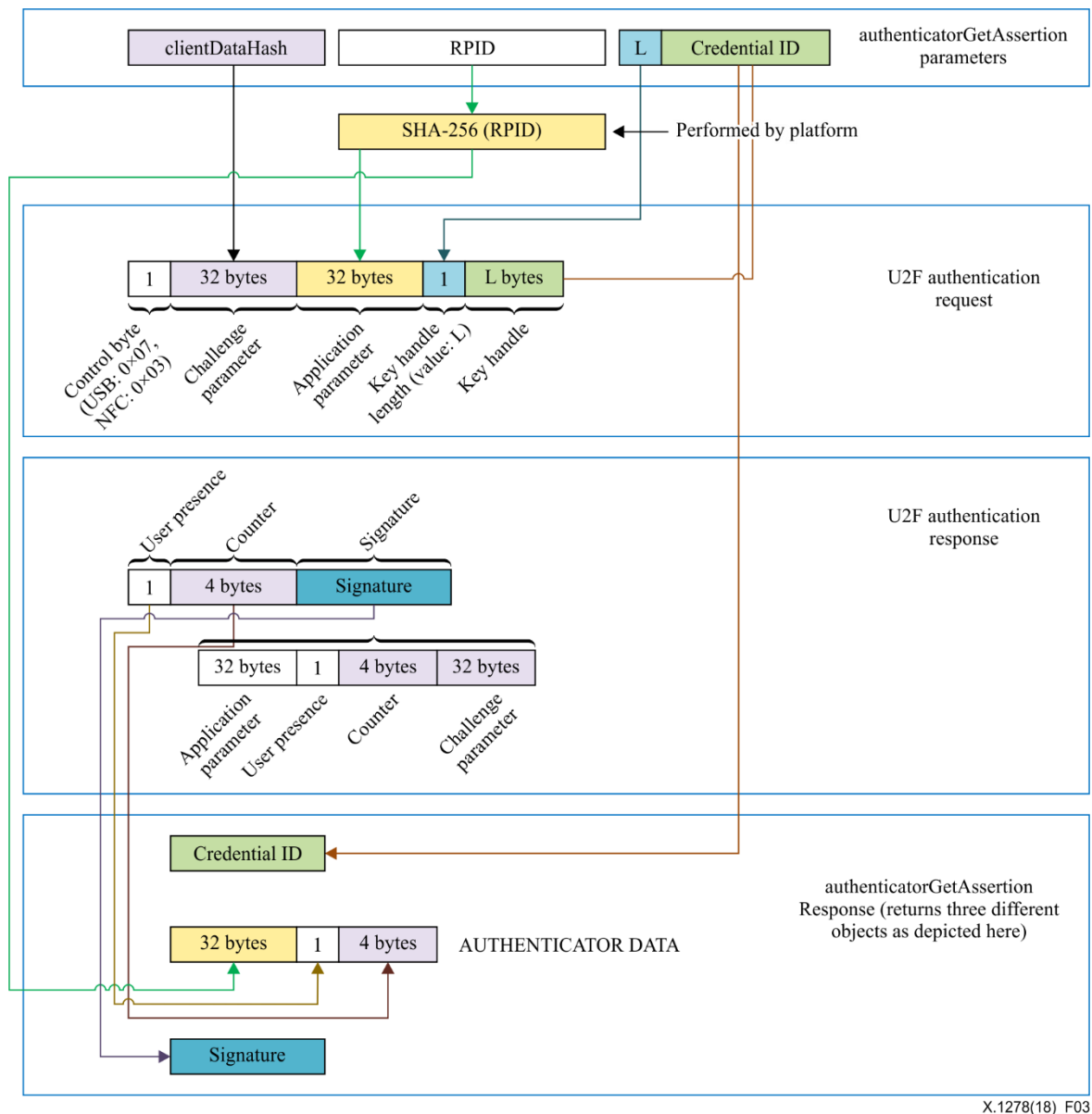
Mapped CTAP2 authenticatorGetAssertion response(CBOR)

```
{1: {"type": "public-key",
      "id": h'3EBD89BF77EC509755EE9C2635EFAAAC7B2B9C5CEF1736C3717DA48534C8C6B6
```

```

54D7FF945F50B5CC4E78055BDD396B64F78DA2C5F96200CCD415CD08FE420038'}},
2: h'1194228DA8FDBDEEFD261BD7B6595CFD70A50D70C6407BCF013DE96D4EFB17DE
010000003B',
3: h'304402207BDE0A52AC1F4C8B27E003A370CD66A4C7118DD22D5447835F45B99C
68423FF702203C517B47877F85782DE10086A783D1E7DF4E3639E771F5F6AFA3
5AAD5373858E'}

```



X.1278(18)_F03

Figure 3 – Mapping: WebAuthn authenticatorGetAssertion to and from CTAP1/U2F authentication messages

13 Transport-specific bindings

13.1 USB

13.1.1 Design rationale

CTAP (CTAP1/U2F and CTAP2) messages are framed for USB transport using the human interface device (HID) protocol. We henceforth refer to the protocol as CTAPHID. The CTAPHID protocol is designed with the following design objectives in mind:

- driverless 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.

13.1.2 Protocol structure and data framing

The CTAP (CTAP1/U2F and CTAP2) protocol is designed to be concurrent and stateless 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 before a second transaction can take place and a response is never sent without a previous request. Transactions exist only at the highest CTAP (CTAP1/U2F and CTAP2) 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.

13.1.3 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 device to ensure its system-wide uniqueness. The actual algorithm for generation of channel identifiers is vendor specific and not defined by this Recommendation.

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

13.1.4 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.

Packets are always of a 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 always has 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.

13.1.5 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) behaving politely, i.e., it does not actively act to destroy 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.

13.1.5.1 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 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. 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.

13.1.5.2 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.

13.1.5.3 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 an idle state to become ready for a new transaction.

13.1.5.4 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.

13.1.6 Channel locking

In order to deal with aggregated transactions that may not be interrupted, such as the tunnelling 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 been considered by general CTAP (CTAP1/U2F and CTAP2) HID applications.

13.1.7 Protocol version and compatibility

The CTAPHID protocol is designed to be extensible, yet maintaining 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.

13.1.8 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 Recommendation. This Recommendation 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 for use given the 8-byte report size limitation together with the protocol overhead.

13.1.8.1 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.

13.1.8.2 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 only provides two "raw" reports, which 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.

13.1.9 CTAPHID commands

The CTAPHID protocol implements the following commands.

13.1.9.1 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 clauses of this Recommendation.

13.1.9.1.1 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 [U2FRawMsgs]. It should be noted that keep-alive messages may be sent from the device to the client before the response message is returned.

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

13.1.9.1.2 CTAPHID_CBOR (0x10)

This command sends an encapsulated CTAP CBOR encoded message. The semantics of the data message is defined in clause 11.

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

13.1.9.1.3 CTAPHID_INIT (0x06)

This command has two functions.

If sent on an allocated channel identifier (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 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 one received. 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. 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 to this current version, but additional fields may be added.

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

The following device capability flags are defined. Unused values are reserved for future use and must be set to zero by device vendors.

CAPABILITY_WINK	If set to 1, authenticator implements CTAPHID_WINK function
CAPABILITY_CBOR	If set to 1, authenticator implements CTAPHID_CBOR function
CAPABILITY_NMSG	If set to 1, authenticator DOES NOT implement CTAPHID_MSG function

13.1.9.1.4 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

13.1.9.1.5 CTAPHID_CANCEL (0x11)

Cancel any outstanding requests on this CID.

Request

CMD	CTAPHID_CANCEL
BCNT	0

Response at success

CMD	CTAPHID_CANCEL
BCNT	0

13.1.9.1.6 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	The command in the request is invalid.
ERR_INVALID_PAR	The parameter(s) in the request is invalid.
ERR_INVALID_LEN	The length field (BCNT) is invalid for the request.
ERR_INVALID_SEQ	The sequence does not match expected value.
ERR_MSG_TIMEOUT	The message has timed out.
ERR_CHANNEL_BUSY	The device is busy for the requesting channel.

13.1.9.1.7 CTAPHID_KEEPLIVE (0x3B)

This command code is sent while processing a CTAPHID_MSG. It should be sent at least every 100 ms and whenever the status changes.

CMD	CTAPHID_KEEPLIVE
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.

13.1.9.2 Optional commands

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

13.1.9.2.1 CTAPHID_WINK (0x08)

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

13.1.9.2.2 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

13.1.9.3 Vendor specific commands

A CTAPHID may implement additional vendor-specific commands that are not defined in this Recommendation, even if they are CTAPHID compliant. Such commands, if implemented must have a command in the range between CTAPHID_VENDOR_FIRST (0x40) and CTAPHID_VENDOR_LAST (0x7F).

13.2 ISO7816, ISO14443 and near-field communication (NFC)

13.2.1 Conformance

Please refer to [b-ISOIEC-7816-4] for application protocol data unit (APDU) definition.

13.2.2 Protocol

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

- 1) The client sends an applet selection command.
- 2) The authenticator replies with success if the applet is present.
- 3) The client sends a command for an operation.
- 4) The authenticator replies with response data or error.

13.2.3 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
AC	0x2f
AX	0x0001

The command to select the FIDO applet is:

CLA	INS	P1	P2	Lc	Data In	Le
0x00	0xA4	0x04	0x0C	0x08	AID	TBD (version string length)

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

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 to maintain backwards compatibility with CTAP1/U2F-only clients.
- If the authenticator only implements CTAP2, the device shall respond with data that is NOT U2F_V2.
- If the authenticator implements both CTAP1/U2F and CTAP2, the version information shall be the string U2F_V2 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.

13.2.4 Framing

Conceptually, framing defines an encapsulation of FIDO2 commands. In NFC, this encapsulation is done in an APDU following [b-ISOIEC-7816-4]. Fragmentation, if needed, is discussed in the following paragraph.

13.2.4.1 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

13.2.4.2 Response

Response shall have the following format in case of success:

Case	Data	Status word
Success	Response data	"9000" – Success
Status update	Status data	"9100" – OK When receiving this, CTAP (CTAP1/U2F and CTAP2) will immediately issue an NFCCTAP_GETRESPONSE command unless a cancel was issued. CTAP (CTAP1/U2F and CTAP2) will provide the status data to the higher layers.
Errors		See [b-ISOIEC-7816-4]

13.2.5 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 an 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.

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.

13.2.6 Commands

13.2.6.1 NFCCTAP_MSG (0x10)

The NFCCTAP_MSG command sends a CTAP (CTAP1/U2F and CTAP2) 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.

13.2.6.2 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.

13.2.7 Bluetooth smart/Bluetooth low energy technology

13.2.7.1 Conformance

The authenticator and client devices using Bluetooth Low Energy technology shall conform to Bluetooth Core Specification 4.0 or later [b-BTCORE]

Bluetooth SIG specified universally unique identifier (UUID) values shall be found on the Assigned Numbers website [b-BTASSNUM].

13.2.7.2 Pairing

Bluetooth low energy technology is a long-range wireless protocol and thus has several implications for privacy, security and overall user-experience. As 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. The authenticator must never use short-term keys.

Bearing in mind that 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 the authenticator 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, the authenticator manufacturers should instruct users to avoid performing Bluetooth pairing in a public space such as a cafe, shop or train station.

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 who 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 clause 13.2.7.12, 'Implementation considerations'.

13.2.7.3 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.

13.2.7.4 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 then discussed for each type of transport layer.

13.2.7.4.1 Request from client to authenticator

Request frames must have the format given in Table 22.

Table 22 – Request frames 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 the commands are described below.

The **CANCEL** command cancels any outstanding **MSG** commands.

The data format for the **MSG** command is defined in clause 11.

13.2.7.4.2 Response from authenticator to client

Response frames must have the format given in Table 23, which share a similar format to the request frames:

Table 23 – Response frames format

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 clause 11.

13.2.7.4.3 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

Error constant	Value	Meaning
ERR_INVALID_SEQ	0x04	The sequence number is invalid
ERR_REQ_TIMEOUT	0x05	The request timed out
NA	0x06	Value reserved (HID)
NA	0x0a	Value reserved (HID)
NA	0x0b	Value reserved (HID)
ERR_OTHER	0x7f	Other, unspecified error

13.2.7.4.5 GATT service description

This profile defines two roles: FIDO authenticator and FIDO client.

- The FIDO client shall be a generic attribute profile (GATT) client.
- The FIDO authenticator shall be a GATT server.

Figure 4 illustrates the mandatory services and characteristics that shall be offered by a FIDO authenticator as part of its GATT server.

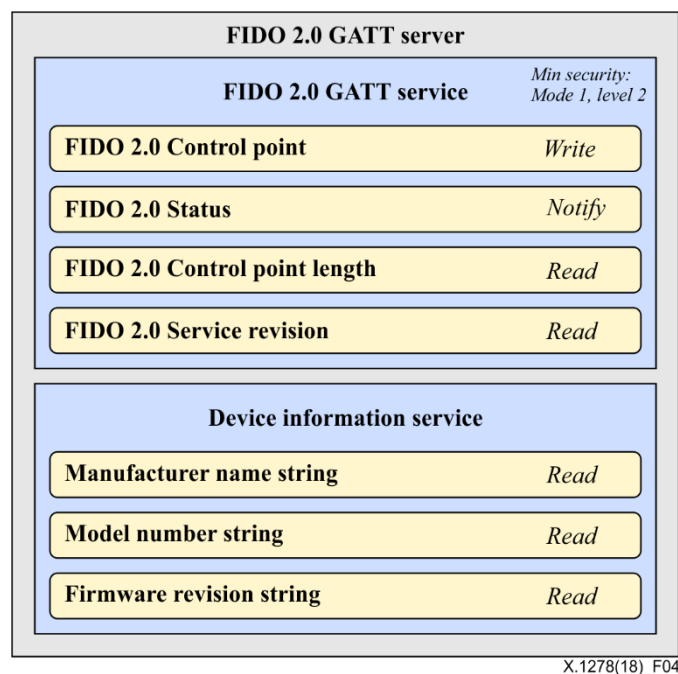


Figure 4 – Mandatory GATT services and characteristics that must be offered by a FIDO authenticator

Note that the generic access service [b-BTGAS] is not present as it is already mandatory for any Bluetooth low energy technology compliant device.

Table 24 summarizes additional GATT sub-procedure requirements for a FIDO authenticator (GATT server) beyond those required by all GATT servers.

Table 24 – GATT sub-procedure requirements for a FIDO authenticator

GATT sub-procedure	Requirements
Write Characteristic Value	Mandatory
Notifications	Mandatory
Read Characteristic Descriptors	Mandatory
Write Characteristic Descriptors	Mandatory

Table 25 summarizes additional GATT sub-procedure requirements for a FIDO client (GATT client) beyond those required by all GATT clients.

Table 25 – GATT sub-procedure requirements for a FIDO client

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.

The 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.

13.2.7.5.1 FIDO service

An authenticator shall implement the FIDO service described in Table 26. The UUID for the FIDO GATT service is 0xFFFD and it shall be declared as a primary service. The service contains the following characteristics:

Table 26 – FIDO services

Characteristic name	Mnemonic	Property	Length	UUID
FIDO Control Point	<code>fidoControlPoint</code>	Write	Defined by vendor (20-512 bytes)	F1D0FFF1-DEAA-ECEE-B42F-C9BA7ED623BB
FIDO Status	<code>fidoStatus</code>	Notify	N/A	F1D0FFF2-DEAA-ECEE-B42F-C9BA7ED623BB
FIDO Control Point Length	<code>fidoControlPointLength</code>	Read	2 bytes	F1D0FFF3-DEAA-ECEE-B42F-C9BA7ED623BB
FIDO Service Revision Bitfield	<code>fidoServiceRevisionBitfield</code>	Read/Write	Defined by vendor (1+ bytes)	F1D0FFF4-DEAA-ECEE-B42F-C9BA7ED623BB
FIDO Service Revision	<code>fidoServiceRevision</code>	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 with 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 also 0. 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	FIDO 2.0
0	4-0	Reserved

For example, a device that only supports FIDO2 Rev 1 will only have a `fidoServiceRevisionBitfield` characteristic of length 1 with the value 0x20.

13.2.7.5.2 Device information service

An authenticator shall implement the device information service [b-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.

13.2.7.5.3 Generic access profile service

Every authenticator shall implement the generic access profile service [b-BTGAS] with the following characteristics:

- Device name
- Appearance

13.2.7.6 Protocol overview

The general overview of the communication protocol follows:

- 1) The authenticator advertises the FIDO service.
- 2) The client scans for the authenticator advertising the FIDO service.
- 3) The client performs characteristic discovery on the authenticator.
- 4) If not already paired, the client and the authenticator shall perform BLE pairing and create an LTK. The authenticator shall only allow connections from previously bonded clients without user intervention.
- 5) The 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) The client reads the `fidoControlPointLength` characteristic.
- 7) The client registers for notifications on the `fidoStatus` characteristic.
- 8) The client writes a request (e.g., an enroll request) into the `fidoControlPoint` characteristic.
- 9) The authenticator evaluates the request and responds by sending notifications over `fidoStatus` characteristic.
- 10) 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.

13.2.7.7 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 [b-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 the 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 [b-BTGAP] and device information service [b-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.

13.2.7.8 Requests

Clients should make requests by connecting to the authenticator and performing a write into the `fidoControlPoint` characteristic.

13.2.7.9 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 the processing of 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 as 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.

13.2.7.10 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 currently not 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 outlined in Table 27.

Table 27 – Initialization fragment

Offset	Length	Mnemonic	Description
0	1	<code>CMD</code>	Command identifier
1	1	<code>HLEN</code>	High part of data length
2	1	<code>LEN</code>	Low part of data length
3	0 to (maxLen – 3)	<code>DATA</code>	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 as defined in Table 28,

Table 28 – Continuation fragment

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 wrap around to 0 after reaching the maximum sequence number of 0x7f.

The following is the 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]

The following is the 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]

13.2.7.11 Notifications

A client needs to register for notifications before it can receive them. Bluetooth Core Specification 4.0 or later [b-BTCORE] forces a device to remember the notification registration status over different connections [b-BTCCC]. Unless a client explicitly unregisters for notifications, the registration will be automatically restored when reconnecting. A client may therefore check the notification status upon connection and only register if notifications are not 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.

13.2.7.12 Implementation considerations

13.2.7.12.1 Bluetooth pairing: client considerations

As outlined in clause 13.2.7.2, 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 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 poses both security and privacy risks to users.

While the client operating system security is partly out of FIDO's scope, further revisions of this Recommendation may propose mitigations for this issue.

13.2.7.12.2 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 so 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 attempting to use an authenticator for the first time with a new client. The client turns on the authenticator, but forgets to put it 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 [b-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 bytes following 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 [b-BTPESTK].

13.2.7.13 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 while anticipating more commands.

A further consideration is to ensure that users are not confused about the command they are 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 (Table 29).
- 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 that 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 that 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 (Table 38) have elapsed without further commands from a client, an authenticator may reset its state or power down.

Table 29 – Timeout values

Constant	Value
<code>kMaxCommandTransmitDelayMillis</code>	1500 milliseconds
<code>kErrorWaitMillis</code>	2000 milliseconds
<code>kKeepAliveMillis</code>	500 milliseconds

13.2.7.14 Data throughput

Bluetooth low energy technology does not have particularly high throughput, this can cause noticeable latency to the user if requests/responses are large. Some ways that implementers can reduce latency are to:

- support the maximum maximum transmission unit (MTU) size allowable by hardware (up to the 512-byte max from the BLE specifications);
- make the attestation certificate as small as possible while excluding unnecessary extensions in the process.

13.2.7.15 Advertising

Though the Recommendation does not appear to mandate it (in any way found thus far), advertising and device discovery seems to work better when the authenticators advertise on all three advertising channels and not just one.

13.2.7.16 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.

Bibliography

- [b-BTASSNUM] Bluetooth Assigned Numbers <<https://www.bluetooth.org/en-us/specification/assigned-numbers>>
- [b-BTCCC] Client Characteristic Configuration. Bluetooth Core Specification 4.0, Volume 3, Part G, Section 3.3.3.3. <<https://www.bluetooth.com/specifications/adopted-specifications>>
- [b-BTCORE] Bluetooth Core Specification 4.0.
<<https://www.bluetooth.com/specifications/adopted-specifications>>
- [b-BTDIS] Device Information Service v1.1.
<https://www.bluetooth.com/specifications/adopted-specifications>
- [b-BTGAP] Generic Access Profile. Bluetooth Core Specification 4.0, Volume 3, Part C, Section 12.
<https://www.bluetooth.com/specifications/adopted-specifications>
- [b-BTGAS] Generic Access Profile service. Bluetooth Core Specification 4.0, Volume 3, Part C, Section 12.
https://developer.bluetooth.org/gatt/services/Pages/ServiceViewer.aspx?u=org.bluetooth.service.generic_access.xml
- [b-BTPESTK] Passkey Entry. Bluetooth Core Specification 4.0, Volume 3, Part H, Section 2.3.5.3. <https://www.bluetooth.com/specifications/adopted-specifications>
- [b-BTSD] Bluetooth Service Data AD Type. Bluetooth Core Specification 4.0, Volume 3, Part C, Section 11. <https://www.bluetooth.com/specifications/adopted-specifications>
- [b-BTXPLAD] Bluetooth TX Power AD Type. Bluetooth Core Specification 4.0, Volume 3, Part C, Section 11. <https://www.bluetooth.com/specifications/adopted-specifications>
- [b-FIDOSrvrGuid] FIDO2 Server Guidelines.
<<https://drafts.fidoalliance.org/fido-2/latest/fido-server-v2.0-wd-20180202.html>>
- [b-IANA-COSE] Schaad, J., Selander, G., Atkins, D., Turner, S. IANA CBOR Object Signing and Encryption (COSE) Algorithms Registry.
<<https://www.iana.org/assignments/cose/cose.xhtml#algorithms>>
- [b-ISOIEC-7816-4] ISO/IEC 7816-4:2013, *Identification cards – Integrated circuit cards– Part 4: Organization, security and commands for interchange.*
- [b-RFC 6090] IETF RFC 6090 (2011), *Fundamental Elliptic Curve Cryptography Algorithms*. <<https://tools.ietf.org/html/rfc6090>>
- [b-RFC 8152] IETF RFC 8152 (2017), *CBOR Object Signing and Encryption (COSE)*.
<<https://tools.ietf.org/html/rfc8152>>
- [b-SP800-56A] [NIST Special Publication 800-56A \(Revised\) \(2007\), Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography](#) <https://csrc.nist.gov/publications/nistpubs/800-56A/SP800-56A_Revision1_Mar08-2007.pdf>
- [b-U2FUsbHid] Fido Alliance Proposed Standard (2017), *FIDO U2F HID Protocol Specification*.
<<https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-hid-protocol-v1.2-ps-20170411.html>>
- [b-U2FNfc] Fido Alliance Proposed Standard (2017), *FIDO NFC Protocol Specification v1.0*.
<<https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-nfc-protocol-v1.2-ps-20170411.html>>
- [b-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>>

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series D	Tariff and accounting principles and international telecommunication/ICT economic and policy issues
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Cable networks and transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Environment and ICTs, climate change, e-waste, energy efficiency; construction, installation and protection of cables and other elements of outside plant
Series M	Telecommunication management, including TMN and network maintenance
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling, and associated measurements and tests
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks, open system communications and security
Series Y	Global information infrastructure, Internet protocol aspects, next-generation networks, Internet of Things and smart cities
Series Z	Languages and general software aspects for telecommunication systems